构建安全Java应用的权威经典，5大社区一致鼎力推荐！

华章原创精品

Java加密与解密的艺术

The Art of Encryption and Decryption about Java

梁栋 著

机械工业出版社
China Machine Press

更多资源请访问稀酷客(www.ckook.com)

The Art of Encryption and Decryption about Java

当你在用IM与好友聊天时，当你通过B2C网站购物时，当你用邮件与客户交流时，当你公司的应用服务器与合作伙伴交换商业数据时……你是否考虑过你的数据是否安全？你的隐私是否会被泄露？你的银行卡是否会被盗用？你的竞争对手是否能破解你的敏感数据？

任何一项通过网络交互的数据都有可能是不安全的，而我们却越来越依赖于网络。如果用户密码、聊天消息、银行账号、邮件信息、商业敏感数据等通过明文传输，后果将不堪设想。自己的账号被盗用、隐私成为公共话题、信用卡被人滥用、竞争对手盗用自己的数据……于是，为了确保数据不被侵犯，数据加密与解密技术应在企业应用中都扮演着非常重要的角色。

如果你在思考下面这些问题，也许本书就是你想要的！

- 作为一名系统架构师，如何让你的系统不留有安全隐患？作为一名程序员，如何让你编写的代码没有安全漏洞？
- 为什么密码学是解决一切安全问题的银弹？密码学究竟是怎样一门学科？近千年来，它经历了怎样的发展历程？它是如何延续至今并逐步发展壮大的？
- 博客、论坛、社区、网络聊天、企业级数据交互应用、网银平台等网络应用都无法逃避网络安全问题，如何在合适的环节选用合适的加密算法，从而提高系统的安全性？
- Java 6支持哪些加密算法？如何扩充Java 6尚不支持的加密算法？如何增强系统的安全级别？
- 消息摘要算法和文件校验算法有什么关联？它与普通的循环冗余校验算法有何差别？如何使用消息摘要算法隐蔽敏感信息？
- 为何Base 64算法可以隐蔽敏感信息但却无法真正起到数据加密的作用，而对称加密算法却能轻而易举地起到数据加密的作用？Base 64算法与对称加密算法之间究竟有何关系？
- Sun并没有提供官方的Base 64算法支持，我们又该如何构建该算法？针对Base 64算法，Apache Commons Codec和Bouncy Castle提供了怎样的支持？在其他加密算法中又起到了怎样的作用？
- 对称加密算法已经几乎能胜任所有的加密需求，为何要研制非对称加密算法？对称加密算法究竟有何弊端？非对称加密算法会是对称加密算法的替代者吗？
- 数字签名是手写签名的数字化产物，其算法与消息摘要算法有何关联？为什么这种算法在结合非对称加密算法密钥后就具备了认证身份的作用？
- 对称加密算法和非对称加密算法如何分发密钥，数字证书在其中充当了何种角色？数字证书又是如何发放的？
- 数字证书集多种加密算法于一身，它是如何传递密钥的？又是如何起身份认证作用的？在HTTPS协议中又是如何与SSL/TLS协议相结合构建安全平台的？
- KeyTool和OpenSSL构建的数字证书究竟有何差别？如何在Java中使用这些工具构建的数字证书？
- 基于HTTPS协议的网银平台，堪称安全级别最高的网络应用，更是密码学应用领域最为成功的案例。Java 6提供了完备的HTTPS协议相关的API，如何使用这些API构建固若金汤的HTTPS平台？
- HTTPS协议和SSL/TLS协议是何关系？这些协议与数字证书、加密算法有何关联？如何使用HTTPS协议构建安全的网络应用？
- 单向认证服务和双向认证服务两者之间有何不同？它们与HTTPS协议有何关系？如何运用这两种认证服务保护我们的应用？

上架指导：计算机/程序设计
ISBN 978-7-111-29762-8
9 787111 297628
定价：69.00元

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com
网上购书：www.china-pub.com

更多资源请访问稀酷客(www.ckook.com)

华章原创精品

Java加密与解密的艺术

The Art of Encryption and Decryption about Java

梁栋 著

机械工业出版社
China Machine Press

更多资源请访问稀酷客(www.ckook.com)

本书是Java安全领域的百科全书，密码学领域的权威经典，5大社区一致鼎力推荐。

全书包含3个部分，基础篇对Java企业级应用的安全知识、密码学核心知识、与Java加密相关的API和通过权限文件加强系统安全方面的知识进行了全面的介绍；实践篇不仅对电子邮件传输算法、消息摘要算法、对称加密算法、非对称加密算法、数字签名算法等现今流行的加密算法的原理进行了全面而深入的剖析，而且还结合翔实的范例说明了各种算法的具体应用场景；综合应用篇既细致地讲解了加密技术对数字证书和SSL/TLS协议的应用，又以示例的方式讲解了加密与解密技术在网络中的实际应用，极具实践指导性。

Java开发者将通过本书掌握密码学和Java加密与解密技术的所有细节；系统架构师将通过本书领悟构建安全企业级应用的要义；其他领域的安全工作者也能通过本书一窥加密与解密技术的精髓。

封底无防伪标均为盗版

版权所有，侵权必究

本书法律顾问　北京市展达律师事务所

图书在版编目（CIP）数据

Java加密与解密的艺术/梁栋著. —北京：机械工业出版社，2010.3

ISBN 978-7-111-29762-8

Ⅰ.J…　Ⅱ.梁…　Ⅲ.JAVA语言—保密编码—程序设计　Ⅳ.TP312

中国版本图书馆CIP数据核字（2010）第027465号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）

责任编辑：李俊竹　　陈佳媛

北京京师印务有限公司印刷

2010年4月第1版第1次印刷

186mm×240mm · 29印张

标准书号：ISBN 978-7-111-29762-8

定价：69.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991，88361066

购书热线：（010）68326294，88379649，68995259

投稿热线：（010）88379604

读者信箱：hzjsj@hzbook.com

更多资源请访问稀酷客(www.ckook.com)

在如今这个信息化时代，数据是一切应用的核心和基础，有数据存在的地方就会有安全隐患，而密码学则是解决绝大多数安全问题的银弹。

Java作为全球最受欢迎的编程语言，它的应用遍及企业级应用的各个领域，安全是所有企业级应用中最突出、重要的问题。然而，这些问题从来就不是一种武器就能解决的。消息摘要算法用于数据校验、对称加密算法用于数据加密、非对称加密算法用于密钥交换、数字签名算法用于身份验证，等等。若要构建安全、坚固的Java企业级应用，不仅要深入了解每种算法的原理并将它们综合运用，而且还要悟透Java加密与解密技术的本质。

作者简介

梁 栋 资深Java开发者，有丰富的Spring、Hibernate、iBatis等Java技术的使用和开发经验，擅长Java企业级应用开发；安全技术专家，对Java加密与解密技术有系统深入的研究，实践经验亦非常丰富。他还是一位出色的项目经理，是V8Booker（手机电子书）项目的核心开发团队成员之一，负责核心模块的开发；同时他还在V8NetBank（网银系统）项目中担任项目经理，负责系统的架构和核心模块的开发。

更多资源请访问稀酷客(www.ckook.com)

Praise 赞　誉

作为一名Java开发者，编写安全的代码比编写优雅的代码更重要，因为安全是一切应用的根本。所有Java开发者都应该全面掌握Java加密与解密的技术，尽可能不让你自己编写的代码给别有用心的人留下可乘之机。如果你是一名Java开发者，强烈建议你阅读并收藏本书，它不仅能作为系统学习Java安全知识之用，还可以作为开发时的参考手册。

——Java开发者社区

作为一名架构师，构建系统时首先应该考虑的就是安全问题。如何才能让你构建的系统坚不可摧，没有安全隐患？掌握加密与解密的技术将会让你在进行系统架构时游刃有余。本书可谓是安全领域的权威经典，是所有Java应用架构师的必备参考手册，强烈推荐。

——架构师社区

本书是目前Java加密与解密领域最全面、最详尽、最前沿的著作之一，它将带领你领略Java安全之美。

——Java中文技术网

密码学是人类最伟大的发明创造之一，是一切安全问题的核心和基础。经过几千年的发展，它在很多行业都发挥着至关重要的作用，尤其是IT领域。本书以通俗的语言，详尽的示例对Java加密与解密的技术进行了详细的阐述，近乎完美。

——Spring开发者社区

对于Java企业级应用开发者而言，加密与解密技术是最重要、最关键的技术之一，必须掌握。本书是Java加密与解密领域的百科全书，不仅内容全面、翔实、实践性强，而且不乏深度。

——51CTO技术博客

更多资源请访问稀酷客(www.ckook.com)

前　言 Preface

众所周知，Java EE是目前企业应用中使用最广泛的技术之一，几乎在任何一个领域都能看到Java EE的身影。随着加密与解密算法的发展，Java加密与解密技术不断演进，不断提高着数据的安全性，已成为各大企业应用中一项关键性的技术。

很多企业应用领域的架构师都很关注加密与解密算法在应用中的使用，譬如用户密码加密、网络协议加密等。如何在名目繁多的Java加密与解密技术中选择合适的算法进行企业级应用开发，如何解决Java加密与解密技术开发过程中遇到的各种问题，这成为许多开发者，尤其是架构师关注的焦点问题。然而，国内目前还没有一本书能解决这些问题。本书的作者因工作需要，采用Java加密与解密技术成功构建了企业级网银系统。在开发过程中，作者感受到了Java加密与解密技术的精妙。作者希望把Java加密与解密技术在企业应用开发领域的经验和心得分享给广大读者，提升企业应用的安全性。

本书面向的读者

本书主要适合以下读者：

❑ 所有利用Java进行企业级应用开发的软件工程师

对于企业级应用软件工程师来讲，这将是一次系统的密码学之旅。本书将介绍密码学理论、Java相关算法实现、开源组件包介绍、数字证书与安全协议等相关内容，并配有相关实例为读者提供详尽实现指导，为构建企业级安全应用提供完整的技术支持。

❑ 系统架构师

对于系统架构师来讲，如何使用成熟技术快速构建安全企业应用是安全工作的第一要务。在算法方面，本书详述了Java 6对于密码学算法的相关实现，针对AES算法密钥长度受限问题给出解决办法。同时，针对当前Java 6不支持的算法，如SHA224、ElGamal和ECDSA等，本书详细介绍如何使用第三方开源加密组件包Bouncy Castle进行相关算法实现补充，并且还详细介

更多资源请访问稀酷客(www.ckook.com)

绍了Apache Commons Codec，这些成熟的组件包都是构建安全企业应用必不可少的工具包。在架构方面，本书浓墨重彩地介绍了数字证书的构建、SSL/TLS协议服务搭建，并通过相关实例介绍如何构建单向/双向认证服务。

❑ 其他安全领域的软件工程师

如今企业级应用已经逐步转变以服务为主的异构体应用，如Web Service应用等。Java加密算法实现遵循密码学相关国际标准，完全可以与其他计算机语言（如C++、C#等）构建的异构体应用进行数据加密交互。本书为读者选择合适的算法并提供详尽的技术实现。

如何阅读本书

全书共分为3个部分：基础篇、实践篇、综合应用篇。

❑ 基础篇

本篇共包含4个章节，主要对Java企业级应用安全、密码学理论和Java中与加密相关的API进行了详细介绍，并详细阐述了第三方组件包Bouncy Castle和Apache Commons Codec相关的API。

第1章主要阐述了当前的安全问题，并给出了安全的相关标准。本书将在后续章节内通过各个算法介绍逐一实现这些标准，这些标准也是评判系统安全级别的准则。

第2章主要详述了密码学相关理论知识，并回顾密码学的发展历程。未曾接触过密码学的读者，可通过本章了解密码学理论的基础，本书将在后续章节中多处应用该章相关技术名词。

第3章详细阐述了Java 6安全领域相关API内容，为读者详尽介绍每一个与密码学相关的类以及方法。该章将是每位安全领域软件工程师必读的内容，在阅读本书的后续章节时需经常翻阅该章内容。

第4章主要介绍如何通过权限文件加强系统安全级别，并详述开源组件Bouncy Castle和Apache Commons Codec相关的API内容。如果您正苦于AES算法密钥长度受限，SHA224、ElGamal、ECDSA等算法缺少支持等问题，那么请您阅读该章；如果您非常希望找到Base64及十六进制编码算法的成熟开源组件，也请您阅读该章。本书将在后续章节中介绍如何使用这些开源组件并实现相关算法。

❑ 实践篇

这篇主要对现今流行的所有加密算法进行了全面阐述和深入剖析，并配合相关测试用例演示算法实现。在阅读这篇前，请阅读本书第2章相关理论知识，并了解第3～4章相关的API内容。

更多资源请访问稀酷客(www.ckook.com)

这篇将是所有企业级应用Java软件工程师的必读内容。

第5章介绍了极为简单的Base64算法，该算法可以作为加密算法的入门算法。如果仅仅需要确保应用交互之间的数据达到隐藏的目的，那么您在第5章中一定可以找到满意的答案。

第6章主要详述了MD系列、SHA系列以及MAC系列三大消息摘要算法相关实现。并详细介绍如何使用Bouncy Castle构建Java 6所不支持的算法实现。对于一般网络应用，经常需要为下载软件提供对应的摘要信息用于校验文件完整性。相信在阅读这章内容后，您可以熟练地使用Apcahe Commons Codec为应用实现校验文件完整性的需求。

第7章将沿着对称加密算法的发展历程，详述DES、DESede、AES和PBE四大算法的实现细节。并详细介绍如何使用Bouncy Castle构建目前较为常用的IDEA算法。这些算法适用于中小型企业级应用网络数据加密交互需求，同时也适用于其他安全领域的相关需求，是应用最为广泛的加密算法，更是密码学领域的的核心算法。如果仅仅想要通过对称加密算法以及消息摘要算法构建简单的加密网络应用，那么该章提供的实例将非常合适。

第8章主要详述了构建于对称加密算法之上的非对称加密算法，包括DH、RSA和ElGamal三大常用算法。该章是本书后续章节内容的基础，数字签名算法、数字证书、安全协议等内容都与该章内容息息相关，请在阅读后续章节前能够对该章内容有较深入的阅读。如果对单向/双向认证服务底层实现非常有兴趣，并想要知道它的来龙去脉，那么该章就是探究该技术旅途上的第一个驿站。

第9章详述了基于消息摘要算法和非对称加密算法之上的数字签名算法，包括RSA、DSA和ECDSA三大常用算法。数字签名算法是消息摘要算法的延续，是单向/双向认证服务核心认证技术。如果想通过非对称加密算法构建简单的网络加密应用，并期望使用数字签名算法对数据进行校验，那么该章的实例将非常合适。

❑ 综合应用篇

这篇不仅细致地介绍了加密技术对数字证书和SSL/TLS协议的应用，而且还以示例的方式讲解了加密解密技术在实际网络中的各种应用，极具实践指导性。请在阅读这篇前仔细阅读实践篇的相关内容。这篇内容将是系统架构师的最爱。

第10章详细介绍了如何使用KeyTool和OpenSSL两大工具进行数字证书管理，并详细介绍如何在Java中使用数字证书。数字证书是非对称加密算法公钥的载体，是SSL/TLS协议和单向/双向认证服务基础。如果想要构建安全的HTTPS网络服务应用，请先阅读该章内容。

第11章主要介绍了SSL/TLS协议及单向/双向认证服务。这将是探究单向/双向认证服务技术旅途上的最后一站。该章将详述如何通过简单配置Tomcat服务器快速构建单向/双向认证服

更多资源请访问稀酷客(www.ckook.com)

务，内容详实、极具实践性。

第12章是本书的实例集合，通过三套网络应用实例揭示常规网络应用安全、即时通信网络应用安全和以数据交互为主的Web Service应用安全，并通过网络监测工具WireShark对其效果进行检测。通过不同算法的组合，三套实例逐步升级自身系统的安全级别，极具指导意义。该章为解决网络安全问题提供了可行性参考。

通过阅读本书，读者不仅能全面掌握Java加密与解密的各种基础知识，而且还能进一步了解Java加密与解密的高级技术和技巧，从而将这些知识都运用到实际开发中去。

更多资源请访问稀酷客(www.ckook.com)

目 录 Contents

前 言

第一部分 基础篇

第1章 企业应用安全 ……2

1.1 我们身边的安全问题 ……2
1.2 拿什么来拯救你，我的应用 ……3
1.2.1 安全技术目标 ……3
1.2.2 OSI安全体系结构 ……4
1.2.3 TCP/IP安全体系结构 ……6
1.3 捍卫企业应用安全的银弹 ……8
1.3.1 密码学在安全领域中的身影 …8
1.3.2 密码学与Java EE ……8
1.4 为你的企业应用上把锁 ……9
1.5 小结 ……10

第2章 企业应用安全的银弹——密码学 ……11

2.1 密码学的发家史 ……11
2.1.1 手工加密阶段 ……11
2.1.2 机械加密阶段 ……12
2.1.3 计算机加密阶段 ……13
2.2 密码学定义、术语及其分类 ……15
2.2.1 密码学常用术语 ……15
2.2.2 密码学分类 ……16
2.3 保密通信模型 ……17
2.4 古典密码 ……18
2.5 对称密码体制 ……19
2.5.1 流密码 ……20
2.5.2 分组密码 ……21
2.6 非对称密码体制 ……26
2.7 散列函数 ……28
2.8 数字签名 ……29
2.9 密码学的未来 ……30
2.9.1 密码算法的破解 ……31
2.9.2 密码学的明天 ……31
2.10 小结 ……32

第3章 Java加密利器 ……34

3.1 Java与密码学 ……34
3.1.1 Java安全领域组成部分 ……34
3.1.2 关于出口的限制 ……36
3.1.3 本书所使用的软件 ……36
3.1.4 关于本章内容 ……37
3.2 java.security包详解 ……37
3.2.1 Provider ……38
3.2.2 Security ……41
3.2.3 MessageDigest ……43
3.2.4 DigestInputStream ……46
3.2.5 DigestOutputStream ……47
3.2.6 Key ……49
3.2.7 AlgorithmParameters ……50

更多资源请访问稀酷客(www.ckook.com)

3.2.8 AlgorithmParameter-Generator ……52
3.2.9 KeyPair ……53
3.2.10 KeyPairGenerator ……54
3.2.11 KeyFactory ……56
3.2.12 SecureRandom ……57
3.2.13 Signature ……59
3.2.14 SignedObject ……62
3.2.15 Timestamp ……63
3.2.16 CodeSigner ……64
3.2.17 KeyStore ……66
3.3 javax.crypto包详解 ……70
3.3.1 Mac ……70
3.3.2 KeyGenerator ……72
3.3.3 KeyAgreement ……74
3.3.4 SecretKeyFactory ……75
3.3.5 Cipher ……77
3.3.6 CipherInputStream ……81
3.3.7 CipherOutputStream ……83
3.3.8 SealedObject ……84
3.4 java.security.spec包和 javax.crypto.spec包详解 ……85
3.4.1 KeySpec和Algorithm-ParameterSpec ……85
3.4.2 EncodedKeySpec ……86
3.4.3 SecretKeySpec ……89
3.4.4 DESKeySpec ……90
3.5 java.security.cert包详解 ……91
3.5.1 Certificate ……91
3.5.2 CertificateFactory ……92
3.5.3 X509Certificate ……94
3.5.4 CRL ……95
3.5.5 X509CRLEntry ……96
3.5.6 X509CRL ……97
3.5.7 CertPath ……99
3.6 javax.net.ssl包详解 ……100
3.6.1 KeyManagerFactory ……100
3.6.2 TrustManagerFactory ……101
3.6.3 SSLContext ……103
3.6.4 HttpsURLConnection ……105
3.7 小结 ……107

第4章 他山之石，可以攻玉 ……109

4.1 加固你的系统 ……109
4.1.1 获得权限文件 ……110
4.1.2 配置权限文件 ……110
4.1.3 验证配置 ……111
4.2 加密组件Bouncy Castle ……111
4.2.1 获得加密组件 ……112
4.2.2 扩充算法支持 ……112
4.2.3 相关API ……116
4.3 辅助工具Commons Codec ……120
4.3.1 获得辅助工具 ……120
4.3.2 相关API ……121
4.4 小结 ……131

第二部分 实践篇

第5章 电子邮件传输算法——Base64 ……134

5.1 Base64算法的由来 ……134
5.2 Base64算法的定义 ……134
5.3 Base64算法与加密算法的关系 ……135
5.4 实现原理 ……136
5.4.1 ASCII码字符编码 ……136
5.4.2 非ASCII码字符编码 ……137
5.5 模型分析 ……137

更多资源请访问稀酷客(www.ckook.com)

5.6 Base64算法实现 ……138
5.6.1 Bouncy Castle ……138
5.6.2 Commons Codec ……140
5.6.3 两种实现方式的差异 ……144
5.6.4 不得不说的问题 ……144
5.7 Url Base64算法实现 ……147
5.7.1 Bouncy Castle ……147
5.7.2 Commons Codec ……149
5.7.3 两种实现方式的差异 ……150
5.8 应用举例 ……151
5.8.1 电子邮件传输 ……151
5.8.2 网络数据传输 ……151
5.8.3 密钥存储 ……152
5.8.4 数字证书存储 ……152
5.9 小结 ……153

第6章 验证数据完整性——消息摘要算法 ……155

6.1 消息摘要算法简述 ……155
6.1.1 消息摘要算法的由来 ……155
6.1.2 消息摘要算法的家谱 ……156
6.2 MD算法家族 ……157
6.2.1 简述 ……157
6.2.2 模型分析 ……158
6.2.3 实现 ……160
6.3 SHA算法家族 ……167
6.3.1 简述 ……167
6.3.2 模型分析 ……168
6.3.3 实现 ……169
6.4 MAC算法家族 ……181
6.4.1 简述 ……181
6.4.2 模型分析 ……182
6.4.3 实现 ……182
6.5 其他消息摘要算法 ……195
6.5.1 简述 ……195
6.5.2 实现 ……195
6.6 循环冗余校验算法——CRC算法 ……206
6.6.1 简述 ……207
6.6.2 模型分析 ……207
6.6.3 实现 ……208
6.7 实例：文件校验 ……209
6.8 小结 ……211

第7章 初等数据加密——对称加密算法 ……213

7.1 对称加密算法简述 ……213
7.1.1 对称加密算法的由来 ……213
7.1.2 对称加密算法的家谱 ……214
7.2 数据加密标准——DES ……214
7.2.1 简述 ……214
7.2.2 模型分析 ……215
7.2.3 实现 ……216
7.3 三重DES——DESede ……222
7.3.1 简述 ……222
7.3.2 实现 ……222
7.4 高级数据加密标准——AES ……227
7.4.1 简述 ……227
7.4.2 实现 ……228
7.5 国际数据加密标准——IDEA ……232
7.5.1 简述 ……232
7.5.2 实现 ……232
7.6 基于口令加密——PBE ……236
7.6.1 简述 ……236
7.6.2 模型分析 ……236
7.6.3 实现 ……237
7.7 实例：对称加密网络应用 ……242
7.8 小结 ……254

更多资源请访问稀酷客(www.ckook.com)

第8章 高等数据加密——非对称加密算法 ……256

8.1 非对称加密算法简述 ……256

8.1.1 非对称加密算法的由来 …256

8.1.2 非对称加密算法的家谱 …257

8.2 密钥交换算法——DH ……258

8.2.1 简述 ……258

8.2.2 模型分析 ……258

8.2.3 实现 ……260

8.3 典型非对称加密算法——RSA …269

8.3.1 简述 ……269

8.3.2 模型分析 ……269

8.3.3 实现 ……271

8.4 常用非对称加密算法——ElGamal…277

8.4.1 简述 ……277

8.4.2 模型分析 ……277

8.4.3 实现 ……278

8.5 实例：非对称加密网络应用 ……284

8.6 小结 ……296

第9章 带密钥的消息摘要算法——数字签名算法 ……297

9.1 数字签名算法简述 ……297

9.1.1 数字签名算法的由来 ……297

9.1.2 数字签名算法的家谱 ……298

9.2 模型分析 ……298

9.3 经典数字签名算法——RSA ……299

9.3.1 简述 ……300

9.3.2 实现 ……300

9.4 数字签名标准算法——DSA ……306

9.4.1 简述 ……306

9.4.2 实现 ……306

9.5 椭圆曲线数字签名算法——ECDSA ……311

9.5.1 简述 ……311

9.5.2 实现 ……311

9.6 实例：带有数字签名的加密网络应用 ……318

9.7 小结 ……329

第三部分 综合应用篇

第10章 终极武器——数字证书 …332

10.1 数字证书详解 ……332

10.2 模型分析 ……335

10.2.1 证书签发 ……335

10.2.2 加密交互 ……335

10.3 证书管理 ……337

10.3.1 KeyTool证书管理 ……337

10.3.2 OpenSSL证书管理 ……341

10.4 证书使用 ……351

10.5 应用举例 ……360

10.6 小结 ……360

第11章 终极装备——安全协议 …362

11.1 安全协议简述 ……362

11.1.1 HTTPS协议 ……362

11.1.2 SSL/TLS协议 ……363

11.2 模型分析 ……364

11.2.1 协商算法 ……365

11.2.2 验证证书 ……365

11.2.3 产生密钥 ……366

11.2.4 加密交互 ……368

11.3 单向认证服务 ……369

11.3.1 准备工作 ……369

11.3.2 服务验证 ……374

更多资源请访问稀酷客(www.ckook.com)

11.3.3 代码验证 ……376
11.4 双向认证服务 ……381
11.4.1 准备工作 ……381
11.4.2 服务验证 ……384
11.4.3 代码验证 ……386
11.5 应用举例 ……387
11.6 小结 ……387

第12章 量体裁衣——为应用选择合适的装备 ……389

12.1 实例：常规Web应用开发安全 …389
12.1.1 常规Web应用基本实现…389
12.1.2 安全升级1——摘要处理…394
12.1.3 安全升级2——加盐处理…396
12.2 实例：IM应用开发安全 ……399
12.2.1 IM应用开发基本实现 …399
12.2.2 安全升级1——隐藏数据…412
12.2.3 安全升级2——加密数据…415
12.3 实例：Web Service应用开发安全…420
12.3.1 Web Service应用基本实现 ……420
12.3.2 安全升级1——单向认证服务 ……427
12.3.3 安全升级2——双向认证服务 ……438
12.4 小结 ……443

附录A Java 6支持的算法 ……445

附录B Bouncy Castle支持的算法 ……447

更多资源请访问稀酷客(www.ckook.com)

Part1 第一部分

基 础 篇

第1章 企业应用安全

第2章 企业应用安全的银弹——密码学

第3章 Java加密利器

第4章 他山之石，可以攻玉

更多资源请访问稀酷客(www.ckook.com)

第1章

企业应用安全

当计算机将我们包围、当网络无处不在时，安全问题也成为我们日益关心的问题。我们依赖于网络，同时又受限于网络，而网络本身却是不安全的！如今越来越多的企业应用都架设在网络平台之上，虽然能为用户提供更快捷和便利的服务支持，但这些服务支持也越来越庞大。与此同时，为了满足用户日益增长的服务需求，企业应用不断在如何提供更好的服务支持和更大信息量的传输方面加大技术投入。而与此失衡的是，企业应用的安全性却未能受到足够的重视。单凭用户名和口令鉴别用户身份，继而授权用户使用的方式难以确保数据的安全性。

1.1 我们身边的安全问题

安全，似乎是个问题。但是，我们觉得这个话题似乎不是那么关键！通常情况下，我们为用户提供用户名和口令验证的方式就可以避免这个问题，但这不是最佳答案，因为这样做是远远不够的。安全隐患无处不在，还是先来看看我们所处环境的安全状况吧！

❑ 存储问题

闪存芯片的快速革命使得移动存储行业发生了质的变化，各种数据存储在各种不同的移动存储设备上。当一部优盘塞满了公司的年度报表、下一年企划策略等各种商业机密后，突然不翼而飞时，我们才会猛然惊醒——优盘中的数据没有任何安全措施，甚至连口令都没有！

❑ 通信问题

我们习惯于通过IM工具与好友聊天、交换心情、透漏隐私，甚至通过IM工具与合作公司交换公司私密数据！当你的隐私成为公共话题，当你的公司的商业数据被曝光，你突然发现原来IM工具是不安全的！没错，不管是哪一种IM工具，都在不遗余力地告诫用户聊天信息可能被盗取，“安全提示：不要将银行卡号暴露在您的聊天信息中！”相信大家都不会对这条提示信息感到陌生。

❑ B2C、B2B交易问题

到邮局排队汇款的日子已经一去不复返了，取而代之的是网上银行，轻松地点击一下按钮就能顺利完成转账的操作。网上银行的确为我们的生活带来了便利，但是，如果我们有被钓鱼网站骗取银行卡号和密码的不幸遭遇，现在想起来是不是仍然心有余悸？难道没有一种办法能

更多资源请访问稀酷客(www.ckook.com)

确保我们输入的信息被发送到安全的地方吗？

❑ 服务交互问题

随着大型应用对交互性的需求越来越高，这些应用之间的数据交互也越来越频繁，甚至是大批量、高负荷的数据交互。当你公司的应用通过Web Service接口与合作伙伴交互数据的时候，你该如何确定对方就是你所信赖的合作伙伴呢？你的Web Service接口安全吗？

❑ 移动应用服务问题

3G时代已经来临，在不远的某一天，你将完全可以通过手机完成现在只能通过PC完成的事情。视频聊天、B2C购物、银行转账，等等。3G时代预示着智能手机将无所不能！其实手机也是计算机，只不过它与你熟悉的PC在体积上有较大的差别而已。3G手机一样要通过网络完成你要执行的操作，将平台由PC转换为手机，并不能保证手机平台就能比PC平台有着更高的安全性！

用手机在WAP网站上下载一款软件，是再平常不过的事情了。但是，如何避免用户因不够信任该软件而取消下载呢？下载后，手机如何鉴别这个软件是安全的呢？如何避免发布的软件在被客户成功下载之前被篡改呢？

❑ 内部人为问题

前面列举的问题都来源于外部，我们往往忽略了内部人为问题。现在的企业应用都能为用户提供用户名和口令来确保用户的数据安全，但很多时候用户名和口令在数据库中却一目了然，甚至有的是以明文方式存储的！企业内部任何能访问数据库的员工都能轻而易举地盗取用户的用户名口令，冒充用户的身份完成各种合乎用户行为的操作，侵害用户的利益。企业因此被用户投诉之后，却又找不到任何蛛丝马迹。

当我们的利益受到侵犯时我们才会想起安全问题，安全原来如此重要！一不小心，你的企业应用就会因为数据泄露而丧失良机、引发投诉，甚至是巨额赔款！安全问题关系着企业的生死存亡！

1.2 拿什么来拯救你，我的应用

“拿什么来保护你，我的应用？”这几乎是每一位架构师和安全工作者所关注的问题。看了上面那么多让人不寒而栗的安全问题，免不了让我们心里发怵。道高一尺，魔高一丈，我们先来看看有什么武器可以应对企业应用的安全问题。接下来会讨论安全技术目标、OSI安全体系结构与TCP/IP安全体系结构这三方面的内容。

1.2.1 安全技术目标

国际标准化组织（ISO）对“计算机安全”的定义为：“为数据处理系统建立和采取的技术和管理的安全保护，保护计算机硬件、软件数据不因偶然和恶意的原因而遭到破坏、更改和泄露。”根据美国国家信息基础设施（NII）提供的文献，安全技术目标包含保密性（Confidentiality）、完整性（Integrity）、可用性（Availability）、可靠性（Reliability）和抗否认性（Non-Repudiation）。

更多资源请访问稀酷客(www.ckook.com)

- **保密性**：也称做机密性。保密性确保数据仅能被合法的用户访问，即数据不能被未授权的第三方使用。
- **完整性**：主要确保数据只能由授权方或以授权的方式进行修改，即数据在传输过程中不能被未授权方修改。
- **可用性**：主要确保所有数据仅在适当的时候可以由授权方访问。
- **可靠性**：主要确保系统能在规定条件下、规定时间内、完成规定功能时具有稳定的概率。
- **抗否认性**：也称做抗抵赖性，主要确保发送方与接收方在执行各自操作后，对所做的操作不可否认。

除此之外，计算机网络信息系统的其他安全技术目标还包括：

- **可控性**：主要是对信息及信息系统实施安全监控。
- **可审查性**：主要是通过审计、监控、抗否认性等安全机制，确保数据访问者（包括合法用户、攻击者、破坏者、抵赖者）的行为有证可查，当网络出现安全问题时，提供调查依据和手段。
- **认证（鉴别）**：主要确保数据访问者和信息服务者的身份真实有效。
- **访问控制**：主要确保数据不被非授权方或以未授权方式使用。

安全技术目标制定的主旨在于预防安全隐患的发生。安全技术目标是构建安全体系结构的基础。

1.2.2 OSI安全体系结构

OSI参考模型是由国际标准化组织制定的开放式通信系统互联参考模型（Open System Interconnection Reference Model，OSI/RM）。OSI参考模型包括网络通信、安全服务和安全机制。网络通信共分七层，按照由下至上的次序分别由物理层（Physical Layer）、数据链路层（Data Link Layer）、网络层（Network Layer）、传输层（Transport Layer）、会话层（Session Layer）、表示层（Presentation Layer）和应用层（Application Layer）构成。其中，数据链路层通常简称为链路层。国际标准化组织于1989年在原有网络通信协议七层模型的基础上扩充了OSI参考模型，确立了信息安全体系结构，并于1995年再次在技术上进行了修正。OSI安全体系结构包括五类安全服务以及八类安全机制。

OSI参考模型结构如图1-1所示。

五类安全服务包括认证（鉴别）服务、访问控制服务、数据保密性服务、数据完整性服务和抗否认性服务。

- **认证（鉴别）服务**：在网络交互过程中，对收发双方的身份及数据来源进行验证。
- **访问控制服务**：防止未授权用户非法访问资源，包括用户身份认证和用户权限确认。
- **数据保密性服务**：防止数据在传输过程中被破解、泄露。
- **数据完整性服务**：防止数据在传输过程中被篡改。

更多资源请访问稀酷客(www.ckook.com)

❑ **抗否认性服务**：也称为抗抵赖服务或确认服务。防止发送方与接收方双方在执行各自操作后，否认各自所做的操作。

从上述对安全服务的详细描述中我们不难看出，OSI参考模型安全服务紧扣安全技术目标。

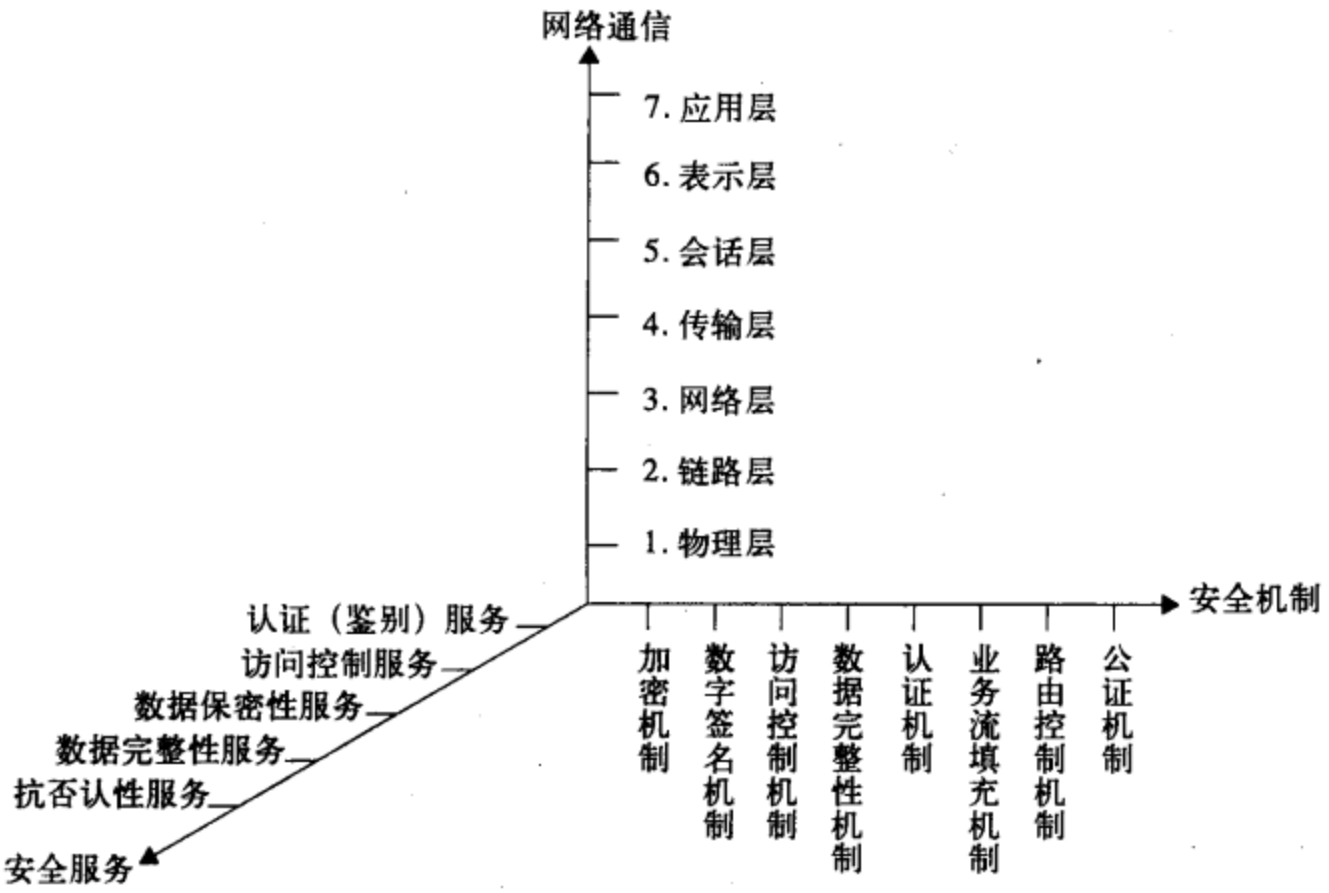


图1-1 OSI参考模型

安全机制是对安全服务的详尽补充。安全服务和安全机制的对应关系如图1-2所示。

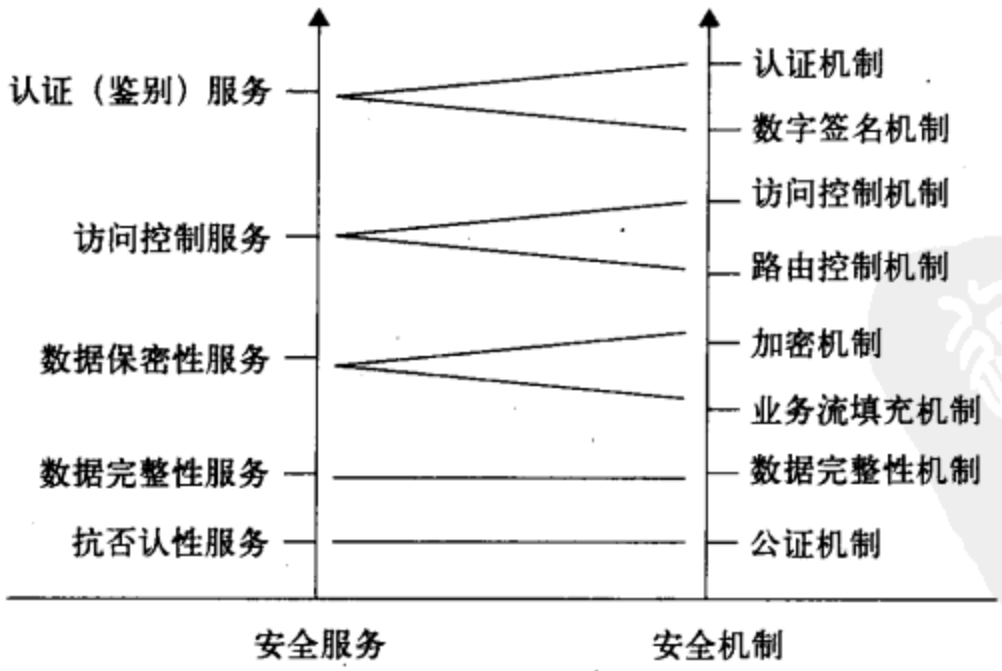


图1-2 OSI参考模型安全服务和安全机制的对应关系

八类安全机制包括加密机制、数字签名机制、访问控制机制、数据完整性机制、认证机制、业务流填充机制、路由控制机制和公证机制。

❑ **加密机制**：加密机制对应数据保密性服务。加密是提高数据安全性的最简便方法。通过对数据进行加密，有效提高了数据的保密性，能防止数据在传输过程中被窃取。常用的

更多资源请访问稀酷客(www.ckook.com)

加密算法有对称加密算法（如DES算法）和非对称加密算法（如RSA算法）。

- **数字签名机制**：数字签名机制对应认证（鉴别）服务。数字签名是有效的鉴别方法，利用数字签名技术可以实施用户身份认证和消息认证，它具有解决收发双方纠纷的能力，是认证（鉴别）服务最核心的技术。在数字签名技术的基础上，为了鉴别软件的有效性，又产生了代码签名技术。常用的签名算法有RSA算法和DSA算法等。
- **访问控制机制**：访问控制机制对应访问控制服务。通过预先设定的规则对用户所访问的数据进行限制。通常，首先是通过用户的用户名和口令进行验证，其次是通过用户角色、用户组等规则进行验证，最后用户才能访问相应的限制资源。一般的应用常使用基于用户角色的访问控制方式，如RBAC（Role Basic Access Control，基于用户角色的访问控制）。
- **数据完整性机制**：数据完整性机制对应数据完整性服务。数据完整性的作用是为了避免数据在传输过程中受到干扰，同时防止数据在传输过程中被篡改，以提高数据传输完整性。通常可以使用单向加密算法对数据加密，生成唯一验证码，用以校验数据完整性。常用的加密算法有MD5算法和SHA算法等。
- **认证机制**：认证机制对应认证（鉴别）服务。认证的目的在于验证接收方所接收到的数据是否来源于所期望的发送方，通常可使用数字签名来进行认证。常用算法有RSA算法和DSA算法等。
- **业务流填充机制**：也称为传输流填充机制。业务流填充机制对应数据保密性服务。业务流填充机制通过在数据传输过程中传送随机数的方式，混淆真实的数据，加大数据破解的难度，提高数据的保密性。
- **路由控制机制**：路由控制机制对应访问控制服务。路由控制机制为数据发送方选择安全网络通信路径，避免发送方使用不安全路径发送数据，提高数据的安全性。
- **公证机制**：公正机制对应抗否认性服务。公证机制的作用在于解决收发双方的纠纷问题，确保两方利益不受损害。类似于现实生活中，合同双方签署合同的同时，需要将合同的第三份交由第三方公证机构进行公证。

安全机制对安全服务做了详尽的补充，针对各种服务选择相应的安全机制可以有效地提高应用安全性。随着技术的不断发展，各项安全机制相关的技术不断提高，尤其是结合加密理论之后，应用安全性得到了显著提高。本书的后续章节将以加密理论及其相应实现为基础，逐步阐述如何通过加密技术确保企业应用的安全。

1.2.3 TCP/IP安全体系结构

OSI参考模型为解决网络问题提供了行之有效的方法，但是卫星和无线网络的出现，使得现有的协议在与卫星和无线网络互联时出现了问题，由此产生了TCP/IP参考模型。TCP/IP从字面上看是两个Internet上的网络协议（TCP是传输控制协议，IP是网际协议），但实际上TCP/IP是一组网络协议，通常包括TCP、IP、UDP、ICMP、RIP、TELNET、FTP、SMTP、ARP、TFTP等协议。TCP/IP参考模型由下至上分为网络接口层、网络层、传输层和应用层。

更多资源请访问稀酷客(www.ckook.com)

OSI参考模型和TCP/IP参考模型的对比如图1-3所示。

OSI参考模型中的物理层和链路层对应TCP/IP参考模型中的网络接口层，网络层和传输层分别对应TCP/IP参考模型中的网络层和传输层，会话层、表示层和应用层对应TCP/IP参考模型中的应用层。

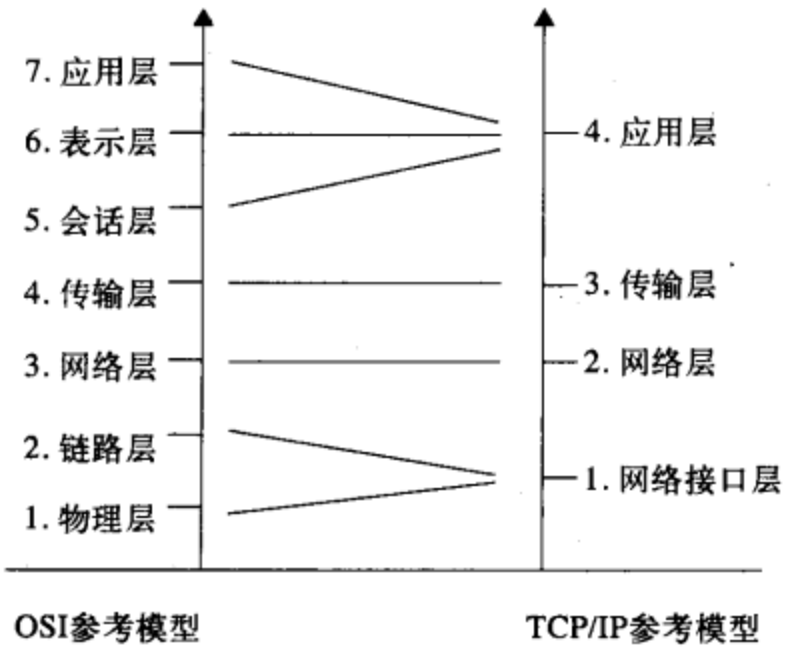


图1-3 OSI参考模型和TCP/IP参考模型对比

对应TCP/IP参考模型，TCP/IP安全体系结构如图1-4所示。

TCP/IP安全体系结构包括网络接口层安全、网络层安全、传输层安全和应用层安全。

- **网络接口层安全**：通常指的是链路层的安全，可以通过加密方式确保数据不被窃听，通常依靠物理层加密实现。一般对通信的链路两端架设加密机，对传输的电器符号进行位流加密。
- **网络层安全**：网络层的功能是负责数据包的路由选择，网络层安全就是要确保数据包能顺利到达指定的目的地。一般通过路由器硬件提高相应的安全性。
- **传输层安全**：传输层的功能是解决端到端的数据传输问题。传输层提供TCP与UDP两种服务：TCP是可靠的、面向连接的服务；UDP是无链接的数据包服务。确保传输层安全有相应的协议，如SSL（Security Socket Layer，安全套接层协议）和TLS（Transport Layer Security，传输层安全协议）。SSL是网景（Netscape）公司设计的主要用于Web的安全传输协议，由IETF（The Internet Engineering Task Force，互联网工程任务组，详见http://www.ietf.org/）将其标准化，进而产生了TLS，TLS是SSL的继任者。SSL 3.0与TLS 1.0差别不大，两种规范大致相同。SSL/TLS协议依赖于加密算法，极难窃听，有较高的安全性。因此，SSL/TLS协议成为网络上最常用的安全保密通信协议，众多电子邮件、网银、电子商务、网上传真都通过SSL/TLS协议确保数据传输安全。随着卫星和无线网络的发展，WAP安全逐渐得到重视。受限于手机及手持设备的处理和存储能力，WAP论坛（http://www.wapforum.org/）在TLS的基础上进行了简化，制定了WTLS协议（Wireless Transport Layer Security，无线传输层安全）。

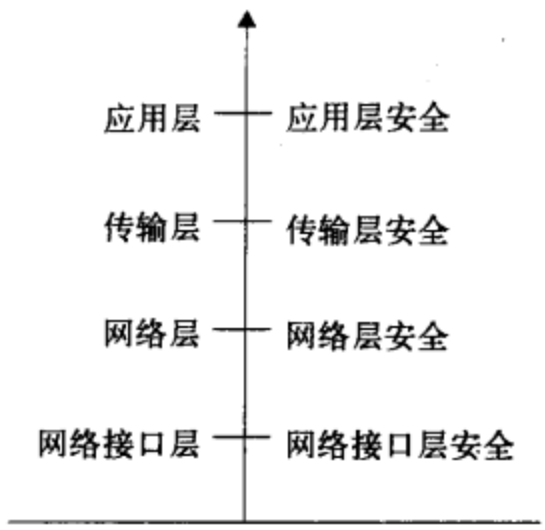


图1-4 TCP/IP安全体系结构

- **应用层安全**：应用层是与应用结合最紧密的一层，负责与应用交互，以实现不同系统的应用之间进行相互通信，完成各种业务处理、提供相应服务。为确保应用层安全可在应用层建立相应的安全机制。HTTPS（Hypertext Transfer Protocol over Secure Socket Layer）协议是Web上最为常用的安全访问协议，简单地说就是HTTP安全版，从HTTPS

更多资源请访问稀酷客(www.ckook.com)

全称不难看出它是基于SSL/TLS的HTTP协议，或者说是HTTPS=SSL/TLS+HTTP。同时，得益于SSL/TLS的高度安全性，HTTPS广泛应用于互联网上的敏感数据交互，例如B2C、网上银行、企业应用之间的数据传递等。本书将在后续章节讲解如何通过HTTPS确保企业应用的安全。

TCP/IP安全体系中，网络接口层安全、网络层安全部分主要通过相应的硬件设施来完成，传输层安全和应用层安全通过HTTPS协议以软件方式完成。

1.3 捍卫企业应用安全的银弹

通过对1.2节的学习，我们已经找到了处理安全问题的武器。但是，我们还缺少一枚解决安全问题的银弹——密码学。的确，密码学是企业应用安全问题领域的一枚银弹，是解决安全问题的核心所在。

1.3.1 密码学在安全领域中的身影

安全领域离不开密码学的支持。例如，在OSI安全体系结构中通过数据加密确保数据的保密性，在TCP/IP安全体系结构中以加密算法为基础构建SSL/TLS协议，这些都说明密码学与安全问题密不可分。

密码学在加密算法上大体可分为单向加密算法、对称加密算法、非对称加密算法三大类。MD5、SHA算法是单向加密算法的代表，单向加密算法是数据完整性验证的常用算法。DES算法是典型的对称加密算法的代表，对称加密算法是数据存储加密的常用算法。RSA算法是典型的非对称加密算法的代表，非对称加密算法是数据传输加密的常用算法。对称加密算法也可以用做数据传输加密，但非对称加密算法在密钥管理方面更有优势。相对对称加密算法而言，非对称加密算法在安全级别上等级更高，但非对称加密算法在时间效率上远不如对称加密算法。

以密码学为基础的各种安全实现相继出现，如HTTPS协议和一系列的“数字技术”（数字摘要、数字信封、数字签名、数字证书等），这些构成了认证技术的基础。

密码学为安全领域筑起了一道铜墙铁壁。

1.3.2 密码学与Java EE

Java EE对密码学的支持是相当广泛的，主要表现在如下几个方面：

- **Java API支持**：Java API支持多种加密算法。如MessageDigest类，可以构建MD5、SHA两种加密算法；Mac类可以构建HMAC加密算法；Cipher类可以构建多种加密算法，如DES、AES、Blowfish对称加密算法，以及RSA、DSA、DH等多种非对称加密算法；Signature类可以用于数字签名和签名验证；Certificate类可用于操作证书；等等。
- **JSP容器支持**：常用的应用服务器（如Tomcat）可以通过简单的配置支持SSL/TLS协议，获取证书配置，有效地构建HTTPS应用。

更多资源请访问稀酷客(www.ckook.com)

- **Java工具支持**：通过KeyTool可以很好地完成密钥管理、证书管理等；通过JarSigner可以完成代码签名。

1.4 为你的企业应用上把锁

终于，我们准备好了应对企业应用安全问题的良策。现在，让我们为自己的企业应用装上这一道道的安全锁。

- **访问控制**：通过为用户设定用户名和口令控制用户访问权限。这是我们最常用的，也是最简单的防范措施。随着企业应用业务的不断细化，如何划分用户访问控制权限，如何控制不同类型的用户（如超级管理员、普通用户、VIP用户）访问受限资源成为新的问题。通常依靠各种理论基础来划分，常用的划分方式有：以用户组为单位划分某组用户可以访问某些资源（Linux操作系统是这种划分方式的典型代表）；以用户角色为单位划分具有何种角色的用户可以访问哪些相应的资源，如我们已经提到过的RBAC。访问控制通常没有固定的算法，由架构师根据系统设计需求进行相应的设计。访问控制仅仅能起到企业应用第一层屏障的作用，最易实现也最不安全，适用于安全系数较低的企业应用。
- **数据加密**：通过对数据的加密、解密可以有效地提高企业应用的安全性。数据加密可以应用在企业应用中的多个环节。例如，可对机要数据进行加密后再存储，对用户的口令加密后存储可有效避免口令盗取导致的用户利益侵犯问题；对网络传数据加密，对用户聊天信息加密传输可以确保用户隐私不易被破译；对要传输的数据做加密摘要，各种通过网络传播的光盘文件（ISO文件）同时附有摘要信息作为验证，可以验证数据完整性。数据加密适用于多种企业应用，架构师可根据具体要求实施相应的加密防范措施。
- **证书认证**：通过数字证书认证可以鉴别用户身份、消息来源的可靠性，加上HTTPS协议的支持可达到高度的数据安全性。数字证书由权威的数字证书认证中心（Certificate Authority，CA)颁布。数字证书经认证中心签名处理，任何第三方都无法修改证书的内容。数字证书自身带有公钥信息，可以对数据进行加密、解密和数字签名验证。同时，带有MD5、SHA的消息摘要信息可做自身有效性验证。数字证书鉴于自身高度的安全性通常以文件形式存储，可通过网络、物理存储载体发送给用户使用。通过读取数字证书信息，HTTPS协议得以发挥安全信道通信的作用，确保数据交互的安全性。当然，安全总是有代价的。通过数字证书对数据做处理后，网络交互时间会相应延迟，主要是因为非对称加密算法的时间效应。但为了更高的安全性，牺牲系统响应时间是有必要的。通常在电子商务中，使用数字证书是最好的选择，也是确保安全交易的唯一选择。而大型企业应用之间的大批量机要数据的交互，通常采用数字证书认证的方式。架构师可根据企业应用的相应领域，不同的业务需求选择有效的数字证书确保企业应用安全。

更多资源请访问稀酷客(www.ckook.com)

1.5 小结

大家都知道安全问题很重要，却不能很好地处理它。每当安全事故发生时，我们才想起要亡羊补牢，但往往为时已晚。在对安全现状做了一些简要总结后，我们发现身边的安全隐患无处不在，PDA里存的年度报表、好友的聊天信息、网上交易的银行卡号以及轻易以明文存储的口令信息等，都可能被盗取、泄露和篡改。

通过安全技术目标的定义，我们知道安全技术目标包含保密性、完整性、可用性、可靠性和抗否认性。这五项技术目标基本上概括了我们所遇到的安全隐患问题，那么我们是否有一整套可行的对策呢？安全体系结构对这个问题给予了相当权威的理论支撑。

OSI参考模型在原有网络通信七层结构的基础上构建了OSI安全体系结构，它由五类安全服务和八类安全机制构成。其中，五类安全服务以安全技术目标为主旨，包括认证（鉴别）服务、访问控制服务、数据保密性服务、数据完整性服务和抗否认性服务；八类安全机制针对五类安全服务做了详尽的补充，包括加密机制、数字签名机制、访问控制机制、数据完整性机制、认证机制、业务流填充机制、路由控制机制和公证机制。

随着卫星和无线网络的出现，原有OSI参考模式已不能应对这些网络的安全问题，TCP/IP得以出现，安全问题也随即突出。为了解决这些安全问题，TCP/IP安全体系结构诞生了，包括网络接口层安全、网络层安全、传输层安全和应用层安全四项内容。其中，网络接口层安全和网络层安全依靠物理硬件来完成，传输层安全和应用层安全通过SSL/TLS+HTTP协议（也就是HTTPS协议）来完成。

通过密码学，我们找到了应对企业应用安全问题的银弹，在Java EE企业应用中也找到了相应的支持。

虽然通过用户名和口令的方式来控制用户访问是最简单，甚至是最简陋的方法，但它仍是企业安全应用的第一道屏障。我们可以合理使用相应的访问控制理论来提高对用户访问控制的安全性。例如基于用户组与用户角色的访问控制方式。在密码学的支持下，我们可以对数据进行加密和解密，确保数据的保密性，也可以通过信息摘要的方式对数据完整性进行验证。单向加密算法可以对数据完整性进行验证，常用算法如MD5、SHA。对称加密算法可以用于数据加密存储，常用算法如DES。非对称加密算法可用于数据加密传输，常用算法如RSA。我们制定出了相应的算法，同时也需要相应的载体。数字证书作为一种凭证，一种载体，可以用于数字加密解密、数字签名验证、自身有效性验证，尤其是当数字证书结合HTTPS协议应用于电子商务后，极大地提高了网络通信安全性。

随着计算机技术的不断发展，新的存储方式、新的网络结构将层出不穷，相应的商务应用环节也会发生翻天覆地的变化，紧随其后的安全问题也会“穷追不舍”。虽然我们可以通过各种技术相应提高企业应用的安全性，但这些技术统统基于密码学理论。同样，密码学并不是固若金汤般的坚不可摧，密码学的破解每一天都有发生。我们应当合理使用各种安全技术，使其有机结合、优势互补，确保企业应用具有更高的安全性。

更多资源请访问稀酷客(www.ckook.com)

第2章

企业应用安全的银弹——密码学

通过对第1章内容的学习，我们基本了解了这枚解决企业应用安全的银弹——密码学。密码学并不是最近才兴起的新学科，它的历史最早可以追溯到几千年以前，而且古今中外都有密码学运用的记载。引用《破译者》书中戴维·卡恩的话，“人类使用密码的历史几乎与使用文字的时间一样长”。在经历战火的洗礼，步入现代以后，计算机科学与数学应用科学的快速发展，促进了密码学的进步。密码学成为应对计算机系统安全问题当之无愧的安全卫士，它广泛应用于计算机与网络安全领域，并逐步进入我们的日常生活：自动柜员机芯片卡、公交IC卡、电子商务，等等。计算机科学的迅速发展，使得密码学在我们的生活中变得越来越重要。

2.1 密码学的发家史

可以这么说，密码学是靠着战争发家的。自从有了战争，就有了密码学的应用环境。在战争中，对阵双方要保护自己的通信安全并窃取、破译对方的情报，于是就有了密码学。用著名的密码学专家罗纳德·李维斯特（Ronald L. Rivest）的话说，“密码学是关于如何在敌人存在的环境中通信”。纵观密码学的发展历程，我们大体可以将其分为三个阶段，即手工加密阶段、机械加密阶段和计算机加密阶段。

2.1.1 手工加密阶段

密码学很早就广泛应用于古代战争中，使用手工方式完成加密操作，以确保战争中军事信息的秘密传送，这一阶段称为手工加密阶段。这一阶段是古典密码学蓬勃发展的时期，称为古典加密阶段。

公元前1000年左右，武王伐纣时期。见于周朝兵书《六韬·龙韬》，书中记载了周朝著名军事家姜子牙为战时通信制定的两种军事通信密码：阴符和阴书。阴符是使用双方在通信前事先制造的一套尺寸不等、形状各异的“阴符”，共八种，每种都代表一定的意义，只有通信双方知道。阴符，可算是密码学中的替代法。阴书在阴符的基础上继续发展，它应用“一合而再离，三发而一知”的理论，也就是密码学中的移位法。将一份完整的军事文书一分为三，分三人传递，必须要把三份文书重新合并后才能获得完整的军事信息。即使途中一人或二人被捕，

更多资源请访问稀酷客(www.ckook.com)

也不至于暴露军事机密。

公元前480年，波希战争。波斯大量军队秘密集结，意图对雅典和斯巴达发动一次突袭。恰逢希腊人狄马拉图斯（Demaratus）在波斯的苏萨城内看到了这次集结，于是他在木板上记录了波斯突袭希腊的意图，然后用蜡把木板上的字封住。这块木板就这样在蜡封的掩盖下送到了希腊，最终使得波斯海军覆没于雅典附近的萨拉米湾——萨拉米湾海战。

公元前404年，斯巴达征服希腊。斯巴达在波斯帝国的帮助下，征服了希腊。斯巴达北路军司令莱萨德还没来得及庆祝就接到密探送来的信函。莱萨德接过密探的腰带，将其缠绕在斯巴达密码棒（Scytale）上，得知波斯帝国意图吞并他的城池。莱萨德当机立断，成功反击了波斯帝国的进攻。斯巴达密码棒实际上是一个指挥棒，将羊皮纸卷在密码棒上，把要保密的信息写在羊皮纸上。由上述“腰带”的含义可知，羊皮纸沿卷轴绕行方向做了切割，切割后的羊皮纸上的信息杂乱无章，信息得以加密。

公元前100年，高卢战争。罗马帝国的凯撒大帝（Caesar）使用以自己的名字命名的密码——凯撒密码，对重要的军事信息进行加密，这是一种简单的单字母替代密码，属于替代法。在当时，凯撒的敌人大多数是目不识丁的，对于这种“移形换位”大法，可谓是根本不知所云。凯撒密码的加密强度，在当时来看是相当有效的。

公元1040年，北宋时期。火药鼻祖曾公亮（999～1078）与北宋文字训诂学家丁度（990～1053）等奉敕集体编撰了《武经总要》，共40卷，该书详细记载了中国古代已知最早的军事情报通信密码。其中，收集了军队中常用的40种战斗情况，编成40条短语，分别编码产生密码本。这套密码的使用方法是，由军事部门指定一首没有重复字的五言律诗（40字），作为解密密钥。诗中的每个字都与短语一一对应，短语顺序在战前临时随机排列。密码本由战时前后两方高级将领保管，前后方通过该密码本进行战时通信。

公元1578年，玛丽女王被伊丽莎白女王软禁，安东尼·贝平顿（Anthony Babington）及其同党意图营救。英国人菲力普·马尼斯（Philip van Marnix）利用频度分析法，成功破解了安东尼发给玛丽的密码信。信件内容除了营救玛丽女王的计划外，还计划行刺伊丽莎白女王。信件的破解将安东尼及其同党一举抓获，审判并处死了玛丽女王。

2.1.2 机械加密阶段

19世纪末至20世纪初，工业革命促进了机械和机电技术的发展，密码学进入机械加密阶段。工业革命为密码学的发展提供了基础，世界大战的爆发为密码学的飞跃提供了契机。

在第一次世界大战中，密码分析有了重大突破，它是战争能否取得胜利的重要因素之一；在第二次世界大战中，密码学经历了它的黄金时代，在战争中扮演了更重要的角色。

1. 第一次世界大战

19世纪末，无线电技术的发明和使用使通信工具发生了革命性的变革。由此产生了以密码技术为核心的包括侦察、测向、信号分析、通信分析等一整套无线电信号侦察以及对抗这种侦察的信号保密技术。军事电报的加密与破解成为同盟国与协约国取得成败的关键。

1914年8月，俄国在芬兰湾口击沉德国“马格德堡”轻巡洋舰，在德国军舰残骸里，俄国

更多资源请访问稀酷客(www.ckook.com)

潜水员意外发现了一份德国海军的密码本，并将其提供给英国，使英国人轻而易举地破译了德国海军的无线电密码。1916年5月30日下午，英国情报部门凭借截获的德国海军无线电密码，破译德国海军电报，日德兰海战以英国皇家海军胜利告终。

1917年1月16日，德意志帝国外交秘书阿瑟·齐默尔曼向德国驻墨西哥大使亨尼希·冯·艾克哈尔特（Heinrich von Eckardt）发出的加密电报——齐默尔曼电报，电报内容建议墨西哥与德意志帝国结成对抗美国的军事联盟。在这个紧要关头，电报内容被英国破译密码的专门机构“40号房间”所截获，利用缴获的德国密码本破译了电报的内容，此次事件被称为“情报史上最伟大的密码破译事件”。“齐默尔曼电报”的破译，促使美国放弃中立参战，改变了战争进程。

2. 第二次世界大战

20世纪初，机械及机电技术的快速发展，加速了密码设备的变革——转轮密码机的发明。转轮密码机的出现是密码学的重要标志之一，促进了传统密码学的进展，提高了机密系统的加密复杂度。转轮密码机Enigma[⊖]，别名“谜”（恩尼格玛密码机）的出现，成为密码学界划时代的丰碑。德国发明家亚瑟·谢尔比乌斯（Arthur Scherbius）发明了Enigma；波兰数学家马里安·雷耶夫斯基（Marian Rejewski）初步破解了简单的Enigma，而英国数学家阿兰·图灵（Alan Turing）[⊜]彻底终结了最高难度的Enigma。

1941年12月8日，美国对日本宣战。在整个太平洋上，美军与日军展开了全面的岛屿争夺战。在电影《风语者》中，日军以成功破解美军军事通信密码，占据战场上的优势，极大地阻碍着美军前进的步伐。1942年，美军征召纳瓦霍人（Navajo，美国最大的印第安部落）加入海军，并训练他们使用纳瓦霍语言作为通信密码。所谓“风语者”，就是使用纳瓦霍语言的通信兵。这是密码学和语言学的成功结合，使得纳瓦霍语成为唯一没有被日本破获的密码，并且成为赢得这场战争的关键。

1942年1月，大西洋海战进入第二阶段。德军以高人一等的密码通信能力，使用“狼群”战术发动大规模无限制潜艇战，致使同盟国节节受挫。在电影《猎杀U-571》中，美军为截获德军密码机在大西洋上与德军潜艇U-571展开了殊死的斗争，最终以截获德军密码机告终。德军密码机的截获，使美军迅速破译了德军指令，扭转了大西洋战事，有力地回击了德军的无限制大规模潜艇战争，加速了第二次世界大战的终结。

2.1.3 计算机加密阶段

第二次世界大战后，计算机与电子学快速发展，促进并推动了密码学进入计算机加密阶段。

⊖ Enigma在密码学界绝对是划时代的丰碑。而且，它所凝聚而成的不是一座丰碑，而是两座：研究并制造出Enigma是一座，研究并破解掉Enigma是另一座。只要稍微了解Enigma的历史，或许很多人就会被其中闪耀的人类智慧之美所折服。如果要向这样辉煌的智慧敬献花环，主要应该献给3个人：首先是德国人亚瑟·谢尔比乌斯，其次是波兰人马里安·雷耶夫斯基，然后是英国人阿兰·图灵。

⊜ 英国数学家、逻辑学家，被称为人工智能之父。1931年图灵进入剑桥大学国王学院，毕业后到美国普林斯顿大学攻读博士学位，第二次世界大战爆发后回到剑桥，后曾协助军方破解德国的著名密码系统Enigma，帮助盟军取得了第二次世界大战的胜利。

更多资源请访问稀酷客(www.ckook.com)

在这一阶段，计算机成为密码设计与破译的平台：利用计算机可以设计出更为复杂的加密算法，避免了徒手设计时容易造成的错误；利用计算机可以对加密算法进行破译，缩短了破译时间。当然，许多设计高明的加密算法的速度通常都很快而且占用资源少，但破解它却需要消耗大量的资源，破解通常以失败告终。在1949年之前，密码学是一门艺术；在1949～1975年，密码学成为科学；1976年以后，密码学有了的新方向——公钥密码学；1977年以后，密码学广泛应用于各种场所。

1949年，信息论始祖克劳德·艾尔伍德·香农（Claude Elwood Shannon）发表了《保密系统的通信理论》一文，把密码学建立在严格的数学基础之上，为密码学的发展奠定了理论基础。密码学由此成为一门真正的科学。在此之前，密码学完全是一门艺术，密码的设计和分析完全依赖于密码专家的直觉。

1976年，密码学专家迪菲（Whitfield Diffie）和赫尔曼（Martin E. Hellman）两人发表了《密码学的新方向》一文，解决了密钥管理的难题，把密钥分为加密的公钥和解密的私钥，提出了密钥交换算法（Diffie-Hellman，D-H），这是密码学的一场革命。

1977年，美国国家标准技术研究所（National Institute of Standards and Technology，NIST）制定数据加密标准（Data Encryption Standard，DES），将其颁布为国家标准，这是密码学历史上一个具有里程碑意义的事件。

同年，密码学专家罗纳德·李维斯特（Ronald L. Rivest）、沙米尔（Adi Shamir）和阿德勒曼（Len Adleman）在美国麻省理工学院，共同提出第一个较完善的公钥密码体制——RSA体制，这是一种建立在大数因子分解基础上的算法，RSA为数字签名奠定了基础。RSA源于整数因子分解问题，DSA源于离散对数问题。RSA和DSA是两种最流行的数字签名机制。数字签名是公钥基础设施（Public Key Infrastructure，PKI）以及许多网络安全机制（SSL/TLS，VPNs等）的基础。自此以后，密码学成为通信、计算机网络、计算机安全等方面的重要工具。

1985年，英国牛津大学物理学家戴维·多伊奇（David Deutsch）提出了量子计算机的初步设想。利用量子计算机，仅需30秒钟即可完成传统计算机要花上100亿年才能完成的大数因子分解，从而破解RSA运用这个大数产生公钥来加密的信息。

同年，物理学家贝内特（Charles H. Bennett）根据多伊奇关于量子密码术的协议，在实验室第一次实现了量子密码加密信息的通信。尽管通信距离仅有30厘米，仍旧证明了量子密码术的实用性。

1997年1月，美国国家标准技术研究所征集新一代数据加密标准，即高级数据加密标准（Advanced Encryption Standard，AES）。最终，比利时密码学家兼计算机科学家Vincent Rijmen和Joan Daemen设计的Rijndael加密算法入选，因此AES算法也称为Rijndael算法。高级数据加密标准用以替代原先的DES，谋求更加安全的加密算法。2002年5月26日，美国国家标准技术研究所将其定为有效的加密标准。

2003年，位于日内瓦的id Quantique公司和位于纽约的MagiQ技术公司，推出了传送量子密钥的距离超越了贝内特实验中30厘米的商业产品。由此，量子密码学进入商业化。

进入计算机加密阶段后，密码学应用不再局限于军事、政治和外交领域，逐步扩大到商务、

更多资源请访问稀酷客(www.ckook.com)

金融和社会的其他各个领域。密码学的研究和应用已大规模扩展到了民用方面。

2.2 密码学定义、术语及其分类

历经四千多年的风风雨雨，密码学逐步发展成为一门学科，对于它的定义也越来越清晰，那么什么是密码学呢？

- 密码学：主要是研究保密通信和信息保密的问题，包括信息保密传输和信息加密存储等。密码学包含密码编码学（Cryptography）和密码分析学（Cryptanalyst）两个分支。编码学与分析学相互促进，又相互制约。一方面，两者在加强密码分析的安全上相互促进；另一方面，两者在实施更为有效的攻击方面也相互影响。
- 密码编码学：主要研究对信息进行编码，实现对信息的隐蔽，是密码学理论的基础，也是保密系统设计的基础。
- 密码分析学：主要研究加密消息的破译或消息的伪造，是检验密码体制安全性最为直接的手段，只有通过实际密码分析考验的密码体制，才是真正可用的。

2.2.1 密码学常用术语

在简要了解了密码学的一些基本概念后，让我们来看一下密码学常用术语，如下所示：

- 明文（Plaintext）：指待加密信息。明文可以是文本文件、图片文件、二进制数据等。
- 密文（Ciphertext）：指经过加密后的明文。密文通常以文本、二进制数据等形式存在。
- 发送者（Sender）：指发送消息的人。
- 接收者（Receiver）：指接收消息的人。
- 加密（Encryption）：指将明文转换为密文的过程。
- 加密算法（Encryption Algorithm）：指将明文变化为密文的转换算法。
- 加密密钥（Encryption Key）：指通过加密算法进行加密操作用的密钥。
- 解密（Decryption）：指将密文转换成明文的过程。
- 解密算法（Decryption Algorithm）：指将密文转换为明文的转换算法。
- 解密密钥（Decryption Key）：指通过解密算法进行解密操作用的密钥。
- 密码分析（Cryptanalysis）：指截获密文者试图通过分析截获的密文从而推断出原来的明文或密钥的过程。
- 密码分析者（Cryptanalyst）：等同于密码破译者，指从事密码分析的人。
- 被动攻击（Passive Attack）：指对一个保密系统采取截获密文并对其进行分析和攻击。这种攻击对密文没有破坏作用。
- 主动攻击（Active Attack）：指攻击者非法入侵密码系统，采用伪造、修改、删除等手段向系统注入假消息进行欺骗。这种攻击对密文具有破坏作用。
- 密码体制（Cipher System）：由明文空间、密文空间、密钥空间、加密算法和解密算法五部分构成。

更多资源请访问稀酷客(www.ckook.com)

- 密码协议（Cryptographic Protocol）：有时也称为安全协议，是指以密码学为基础的消息交换的通信协议，其目的是在网络环境中提供各种安全服务。密码协议与密码算法同等重要，堪称当今密码学研究的两大课题。密码学是网络安全的基础，但网络安全不能单纯依靠安全的密码算法。密码协议是网络安全的一个重要组成部分，通过密码协议进行实体之间的认证、在实体之间安全地分配密钥或其他各种秘密、确认发送和接收的消息的不可否认性等。
- 密码系统（Cryptography）：指用于加密和解密的系统。加密时，系统输入明文和加密密钥，加密变化后，输出密文；解密时，系统输入密文和解密密钥，解密变换后，输入明文。一个密码系统由信源、加密变换、解密变化、信宿和攻击者组成。密码系统强调密码方案的实际应用，通常应当是一个包含软、硬件的系统。
- 柯克霍夫原则（Kerckhoffs' Principle）：数据的安全基于密钥而不是算法的保密。换句话说，系统的安全性取决于密钥，对密钥保密，对算法公开。信息论始祖克劳德·艾尔伍德·香农（Claude Elwood Shannon）将其改为“敌人了解系统”，这样的说法称为香农箴言。柯克霍夫原则是现代密码学设计的基本原则。

2.2.2 密码学分类

密码学起源于古代，发展于现代。随着时间的推移，密码学不断完善，逐步拥有了众多分类。可以从时间上划分，也可以从保密内容的算法上划分，还可以从密码体制上划分，下面将详细介绍这3类密码。

1. 按时间划分

从时间上可以分为古典密码和现代密码，古典密码以字符为基本加密单元，2.4节中将会有详细的阐述；现代密码以信息块为基本加密单元。

2. 按保密内容的算法划分

根据保密内容的算法可分为受限制算法和基于密钥算法。

- 受限制（Restricted）算法：算法的保密性基于保持算法的秘密。一般不赞成使用这种算法，除非应用于类似军事一类的应用，算法由专业机构开发、验证，确保其算法的安全性。这是古典密码学的主要特征。
- 基于密钥（Key-Based）算法：算法的保密性基于对密钥的保密。这其实是基于柯克霍夫原则设计的算法，这样做的好处是：算法的公开有助于算法安全性的验证，算法的漏洞得以及时修正，避免算法的设计者在算法上留下后门等。这正是现代密码学的主要特征。

3. 按密码体制划分

根据密码体制可分为对称密码体制和非对称密码体制。

- 对称密码体制（Symmetric Cryptosystem）：也称为单钥密码体制或私钥密码体制，将在2.5节详细阐述。指该密码体制中的加密密钥与解密密钥相同，即加密过程与解密过程

更多资源请访问稀酷客(www.ckook.com)

使用同一套密钥。

- 非对称密码体制（Asymmetric Cryptosystem）：也称为双钥密码体制或公钥密码体制。指该密码体制中的加密密钥与解密密钥不同，密钥分为公钥与私钥。公钥对外公开，私钥对外保密。

与上述密码体制对应的算法有对称密码算法和非对称密码算法。

- 对称密码算法（Symmetric Cipher）：也称为单钥密码算法或私钥密码算法。指对应于对称密码体制的加密、解密算法。常见的DES、AES算法都是对称密码算法的典范。
- 非对称密码算法（Asymmetric Cipher）：也称为双钥密码算法或公钥密码算法。指对应于非对称密码体制的加密、解密算法。大名鼎鼎的RSA算法就是非对称密码算法，多应用于数字签名、身份认证等。当然，非对称密码算法的相对于对称密码算法有着更高的安全性，却有着不可回避的加密解密的耗时长的问题。

4. 按明文的处理方法划分

根据明文的处理方法可分为分组密码和流密码。

- 分组密码（Block Cipher）：指加密时将明文分成固定长度的组，用同一密钥和算法对每一块加密，输出也是固定长度的密文。分组密码多应用于网络加密。
- 流密码（Stream Cipher）：也称为序列密码。指加密时每次加密一位或一个字节的明文。手机平台对应用使用系统资源有着极为苛刻的要求，这恰恰给了对系统资源要求极低的流密码以用武之地。RC4是相当有名的流密码算法。

在手工加密阶段和机械加密阶段，流密码曾是当时的主流。现代密码学的研究主要关注分组密码和流密码及其应用。在对称密码体制中，大部分加密算法属于分组密码。关于分组密码和流密码的详细内容，请阅读2.5节。

2.3 保密通信模型

密码学并不是孤立地存在的，它需要有一个环境，就是保密通信模型。在了解了密码学的基本术语后，我们来讨论保密通信模型。

密码学的目的在于确保信息的保密传送。通常这样理解这层含义，信息的发送者和接收者在不安全的信道上进行通信，而破译者不能理解他们通信的内容。用保密通信模型来诠释这种信息传送方式，如图2-1所示。

在上述模型中，信息的发送者和接收者要在不安全的信道上交换机要信息，为避免破译者窃听，需要对机要信息进行加密和解密处理。加密信息在传送过程中即使被破译者截获，也不能被破译。基于柯克霍夫原则，只要密钥安全，即便破译者知道该密码系统的加密算法，也无法对加密信息进行破译。在这个模型中，加密密钥很可能和解密密钥是同一个密钥，或者说由一方密钥可以推导出另一方密钥，这就是对称加密密码体制；反之，加密密钥与解密密钥不同，由一方密钥难以推导出另一方密钥，这就是非对称加密密码体制。

更多资源请访问稀酷客(www.ckook.com)

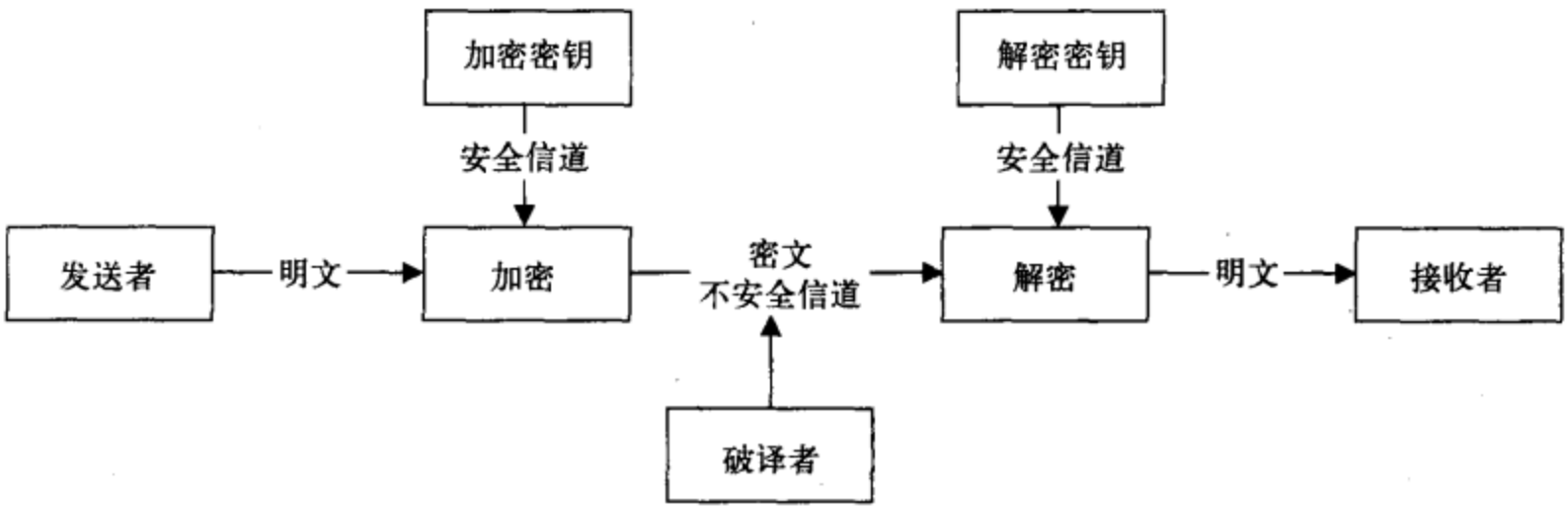


图2-1 保密通信模型

2.4 古典密码

古典密码（Classical Cipher）起始于古代终止于19世纪末，是现代密码的基础。当时生产力水平较低，普遍采用纸、笔或简单器械完成加密、解密操作，正是密码学发展史上手工加密阶段迅速发展的时期。虽然，古典密码由于安全性较低、效率低等多种缺陷已经退出了历史舞台，但古典密码对于密码学的研究却有着不可替代的作用。

古典密码受限于当时的环境，以语言学为基础对文字进行字符变化，也就是对字符加密，以达到信息加密的目的。古典加密算法最常用、最核心的两种加密技巧是移位和替代，这同样是对称加密算法最常用的方法。

- 移位密码（Transposition Cipher）：也称错位密码。将字符的顺序重新排列。例如，将“1234567890”变成“3216549870”。这种加密算法看似简单，但却十分有效。如果不能领会其中的规律，很难辨别其内容的真正含义。
- 替代密码（Substitution Cipher）：也称置换密码。将明文中的一组字符替代成其他的字符，形成密文。例如，“Encryption algorithm”变成“Fodszqujpo bmhpsjuin”（每个字母用下一个字母替代）。接收者对密文做反向替代就可以恢复明文。著名的凯撒密码就应用了替代式算法。

在古典加密时代，替代密码发展迅速，而且变得更加复杂，拥有众多分支，又分为如下几种。

- 单表替代密码（Monoalphabetic Cipher）：也称简单替代密码。明文的一个字符用相应的一个密文字符代替。加密过程中是从明文字母表到密文字母表的一一映射。主要包括移位（shift）密码、乘数（multiplicative）密码、仿射（affine）密码、多项式（Polynomial）密码、密钥短语（Key Word）密码。
- 同音替代密码（Homophonic Substitution Cipher）：也称多名替代密码。与单表替代系统相似，唯一的不同是单个字符明文可以映射成密文的几个字符之一，例如，A可能对应于5、13、25或56，“B”可能对应于7、19、31或42，所以，同音代替的密文并不唯一。电影《风语者》中，美军征召纳瓦霍人加入海军，训练他们使用纳瓦霍语言作为通

更多资源请访问稀酷客(www.ckook.com)

信密码，这实际上是应用了多名替代密码。

- 多表替代密码（Ployalphabetic Substitution Cipher）：明文中的字符映射到密文空间的字符还依赖于它在上下文中的位置。由多个简单的代替密码构成，例如，可能有5个被使用的不同的简单代替密码，单独的一个字符用来改变明文的每个字符的位置。弗吉尼亚（Vigenere）密码、博福特（Beaufort）密码、滚动密钥（running-key）密码、弗纳姆（Vernam）密码、转子机（rotor machine）密码均为多表替代密码。第二次世界大战中，德军用的转子加密机——Enigma，正是多表替代密码应用的典范。
- 多字母替代密码（Polygram Substitution Cipher）：明文中的字符被成组加密，例如“ABA”可能对应于“RTQ”，ABB可能对应于“SLL”等。希尔（Hill）密码、Playfair密码均为多字母替代密码。在第一次世界大战中英国人就采用了这种密码。

不管是移位算法还是替代算法，终究脱离不开人类语言。针对该特点，通过对密文进行语义分析使得古典密码在破译上有章可循。例如，凯撒密码是单表替代密码，要破解凯撒密码，只要以语言学为基础，找出使用频度最高的字符，如' '和 'e'，用ASCII码表示就是32和101，差值69。如果明文中两个出现频率最高的字符的ASCII码相差69，那么加密后密文中相应出现频率最高的字符的ASCII码相差也一定是69。很显然，通过这样的分析方法，只要找出密文中与之对应的字符，计算偏移量——密钥，就可以破译密文，这就是著名的频度分析法。公元1578年，玛丽女王营救计划因密信被破解而以失败告终，当时使用的破解方法就是频度分析法。

2.5 对称密码体制

对称密码体制并不是现代密码学的新生产物，它是古典密码学的进一步延续。古典密码常用的两种技巧——替代和移位，仍然是对称密码体制中最重要的加密技巧。

对称密码体制的保密通信模型如图2-2所示。对称密码体制要求加密与解密使用同一个共享密钥，解密是加密的逆运算，由于通信双方共享同一个密钥，这就要求通信双方必须在通信前商定该密钥，并妥善保存该密钥，该密钥称为秘密密钥。秘密密钥的存在使得对称密码体制开放性变差。

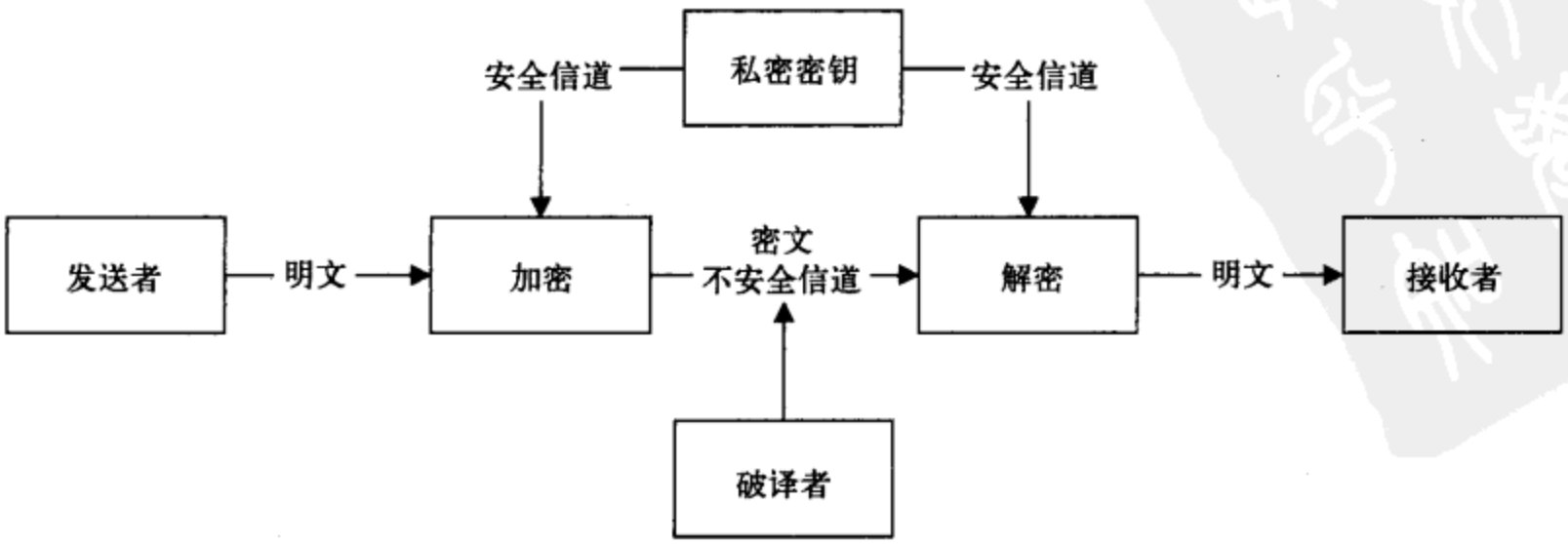


图2-2 对称密码体制的保密通信模型

更多资源请访问稀酷客(www.ckook.com)

对称密码体制分为两种：一种是对明文的单个位（或字节）进行运算，称为流密码，也称为序列密码；另一种是把明文信息划分成不同的组（或块）结构，分别对每个组（或块）进行加密和解密，称为分组密码。

2.5.1 流密码

流密码是军事、外交等机要部门中应用最为广泛的对称密码体制。同时，它也是手机应用平台最常用的加密手段。流密码实现较为简单，加密时将明文按字符（或字节）逐位进行加密，解密时将密文按字符（字节）逐位解密。加密、解密可以是简单的位运算，如模n运算。明文加密后，生成的密文几乎和明文保持同样的长度。流密码加密与解密的流程如图2-3所示。

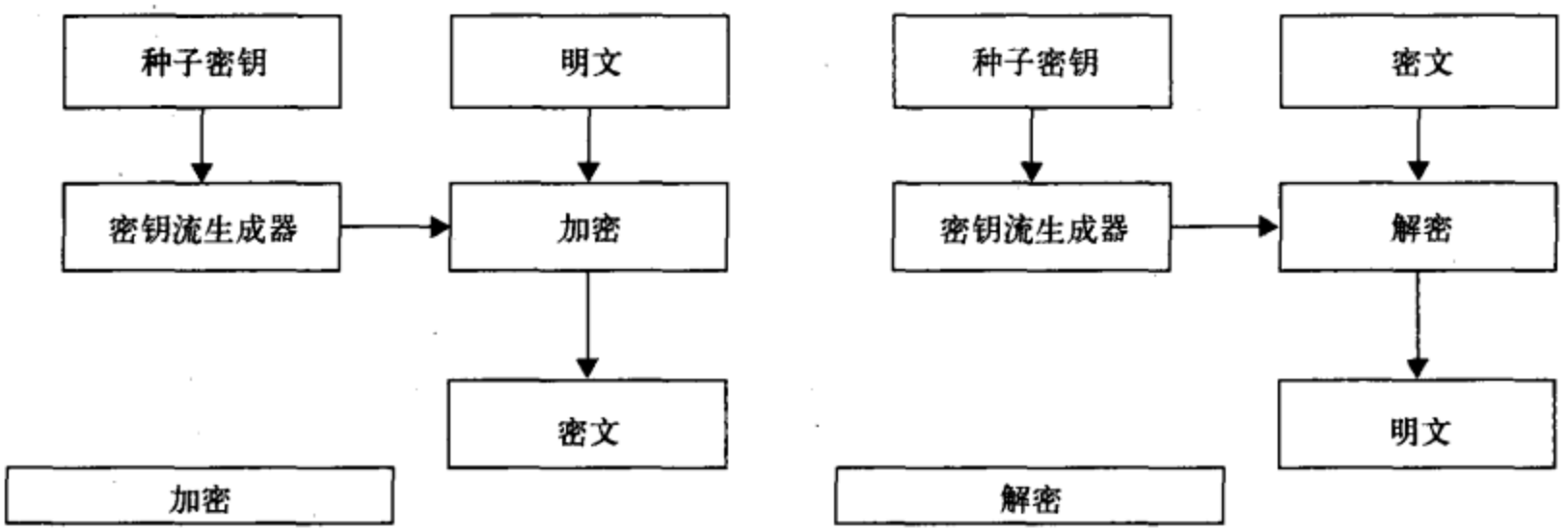


图2-3 流密码加密与解密流程

流密码分为同步流（序列）密码和自同步流（序列）密码。

1. 同步流密码

信息发送方和接收方在传递信息时，同步进行加密解密操作，明文与密文一一对应。密文的内容如果在传输过程中被篡改、删除或插入，可导致同步失效，以致密文解密失败，必须通过重新同步来实现解密、恢复密文。在密文传输过程中，如果一个密文位发生变化，那么该位的变化只影响该位的恢复，对后续密文位不影响，这是同步流密码的一个重要特点。但是，根据该特性主动攻击者可以有选择地对密文字符进行改动，并准确知道这些改动对明文的影响。因此，同步流密码具有同步性、无错误传递性及主动攻击性三种特性。同步流密码适用于为音频和视频数据提供版权保护。

2. 自同步流密码

与同步流密码相比，自同步流密码是一种有记忆变换的密码。每一个密钥与已产生的固定数量的密文位有关，密钥由已生成的密文决定。在密文传输过程中，如果一个密文位发生变化，那么该位的变化会影响到后续有限位（如n位）的密文位的正确解密。所以，自同步流密码有错误传递现象。但是，在接收n位正确密文字符后，密码自身会实现重新同步。基于这一特点，如果主动攻击者对密文做了修改，接收方仍然不能检测出密文的完整性。与同步流密码相比，自同步流密码的密码分析更加困难，安全性更高。因此，自同步流密码具有自同步性、错误传

更多资源请访问稀酷客(www.ckook.com)

递有限性、主动攻击性及明文统计扩散性四种特性。

流密码实现简单、便于硬件计算、加密与解密处理速度快、错误传播率低等优点。但是，流密码对错误的产生不够敏感，这是流密码的缺点。为了弥补这一缺点，流密码通常配合其他技术验证信息的完整性。流密码涉及大量的理论知识，受限于应用场合（目前主要用于军事和外交等机要部门），许多研究成果并未完全公开。目前使用较多的流密码是自同步流密码。流密码的常用算法有RC4和SEAL等。

流密码的安全强度依赖于密钥流生成器所产生的密钥流序列的特征，关键在于密钥生成器的设计以及信息收发两端密钥流产生的同步技术。

2.5.2 分组密码

分组密码多应用于网络加密，是对称密码体制中发展最为完善的密码体制。分组密码对固定长度的一组明文进行加密，这一固定长度称为分组长度。分组长度是分组密码的一个参数，它与分组算法的安全性成正比，其取值范围取决于实际应用的环境。为保证分组算法的安全性，分组长度越长越好，分组长度越长，则密码分析越困难；为保证分组密码的实用性，分组长度越短越好，分组长度越短，则越便于操作和运算。分组长度的设定需要权衡分组算法的安全性与实用性，一般设置为56位。但随着密码学的发展，分组长度只有56位的分组密码已经不能确保算法的安全性。目前，分组密码多选择128位作为算法的分组长度。

分组密码的加密过程是对一个分组长度为n位的明文分组进行加密操作，相应地产生一个n位的密文分组，由此可见，不同的n位明文分组共有2^n个。考虑到加密算法的可逆性（即保证解密过程的可行性），每一个不同的n位明文分组都应该产生一个唯一的密文分组，加密过程对应的变换称为可逆变换或非奇异变换。所以，分组密码算法从本质上来说是定义了一种从分组的明文到相应的密文的可逆变换。

分组密码是现代密码学的重要组成部分，具有代表的分组加密算法有DES、AES等，我们将在后续章节具体探讨如何实现分组密码。

1. 分组密码设计原则

分组密码的设计原则包括安全性和实现性两个方面。前者主要研究如何设计安全算法，分组长度和密钥长度，后者主要讨论如何提高算法的执行速度。

（1）针对安全的一般设计原则

安全性原则也称为不可破译原则，它包含理论上不可破译和实际上不可破译两重含义。香农认为：在理想密码系统中，密文的所有统计特性都与所使用的密钥独立。关于实用密码的两个一般的设计原则是香农提出的混乱原则和扩散原则。

- 扩散（Diffusion）原则：人们所设计的密码应使得密钥的每一位数字影响到密文的多位数字，以防止对密钥进行逐段破译，而且明文的每一位数字也影响密文的多位数字以便隐藏明文数字的统计性。
- 混乱（Confusion）原则：人们所设计的密码应使得密钥和明文以及密文之间的信赖关系相当复杂以至于这种信赖性对密码分析者来说是无法利用的。

更多资源请访问稀酷客(www.ckook.com)

如何衡量一个密码体制的安全性？

主要有以下几个方面：

1）密码体制的破译所需要的时间和费用超出了现有的资源和能力。

2）密码体制的破译所需要的时间超过了该体制所保护的信息的有效时间。

3）密码体制的破译所需要的费用超过了该体制所保护的信息的价值。

（2）针对实现的设计原则

分组密码可以用软件和硬件来实现。硬件实现的优点是可获得高效率，而软件实现的优点是灵活性强、代价低。

- ❑ 软件实现的设计原则：使用子块和简单的运算。密码运算在子块上进行，要求子块的长度能自然地适应软件编程，如8、16、32位等。应尽量避免按位置换，在子块上所进行的密码运算尽量采用易于软件实现的运算。最好是用处理器的基本运算，如加法、乘法、移位等。
- ❑ 硬件实现的设计原则：加密和解密的相似性，即加密和解密过程的不同应局限于密钥使用方式上，以便采用同样的器件来实现加密和解密，以节省费用和体积。尽量采用标准的组件结构，以便能适应于在超大规模集成电路中实现。

2. 分组密码工作模式

我们以DES算法工作模式为例，DES算法根据其加密算法所定义的明文分组的大小（56位），将数据分割成若干56位的加密区块，再以加密区块为单位，分别进行加密处理。如果最后剩下不足一个区块的大小，我们称之为短块，短块的处理方法有填充法、流密码加密法、密文挪用技术。

1980年12月，DES算法工作模式被美国联邦信息处理标准组织（Federal Information Processing Standard，FIPS）标准化。加密算法应用的复杂性，有的强调效率，有的强调安全，有的强调容错性。根据数据加密时每个加密区块间的关联方式来区分，可以分为4种工作模式：电子密码本模式（Electronic Code Book，ECB）、密文链接模式（Cipher Book Chaining，CBC）、密文反馈模式（Cipher Feed Back，CFB）、输出反馈模式（Output Feed Back，OFB）。AES标准除了推荐上述4种工作模式外，还推荐了一种新的工作模式：计数器模式（Counter，CTR）。这些工作模式可适用于各种分组密码算法。

❑ 电子密码本模式——ECB

电子密码本模式如图2-4所示，它是最基本、最易理解的工作模式。每次加密均产生独立的密文分组，每组的加密结果不会对其他分组产生影响，相同的明文加密后对应产生相同的密文，无初始化向量（也称为加密向量）。可以认为有一个非常大的电码本，对任意一个可能的明文分组，电码本中都有一项对应于它的密文，这也是该模式名称的由来。

- 优点：易于理解且简单易行；便于实现并行操作；没有误差传递的问题。
- 缺点：不能隐藏明文的模式，如果明文重复，对应的密文也会重复，密文内容很容易被替换、重排、删除、重放；对明文进行主动攻击的可能性较高。
- 用途：适合加密密钥，随机数等短数据。例如，安全地传递DES密钥，ECB是最合适的模式。

更多资源请访问稀酷客(www.ckook.com)

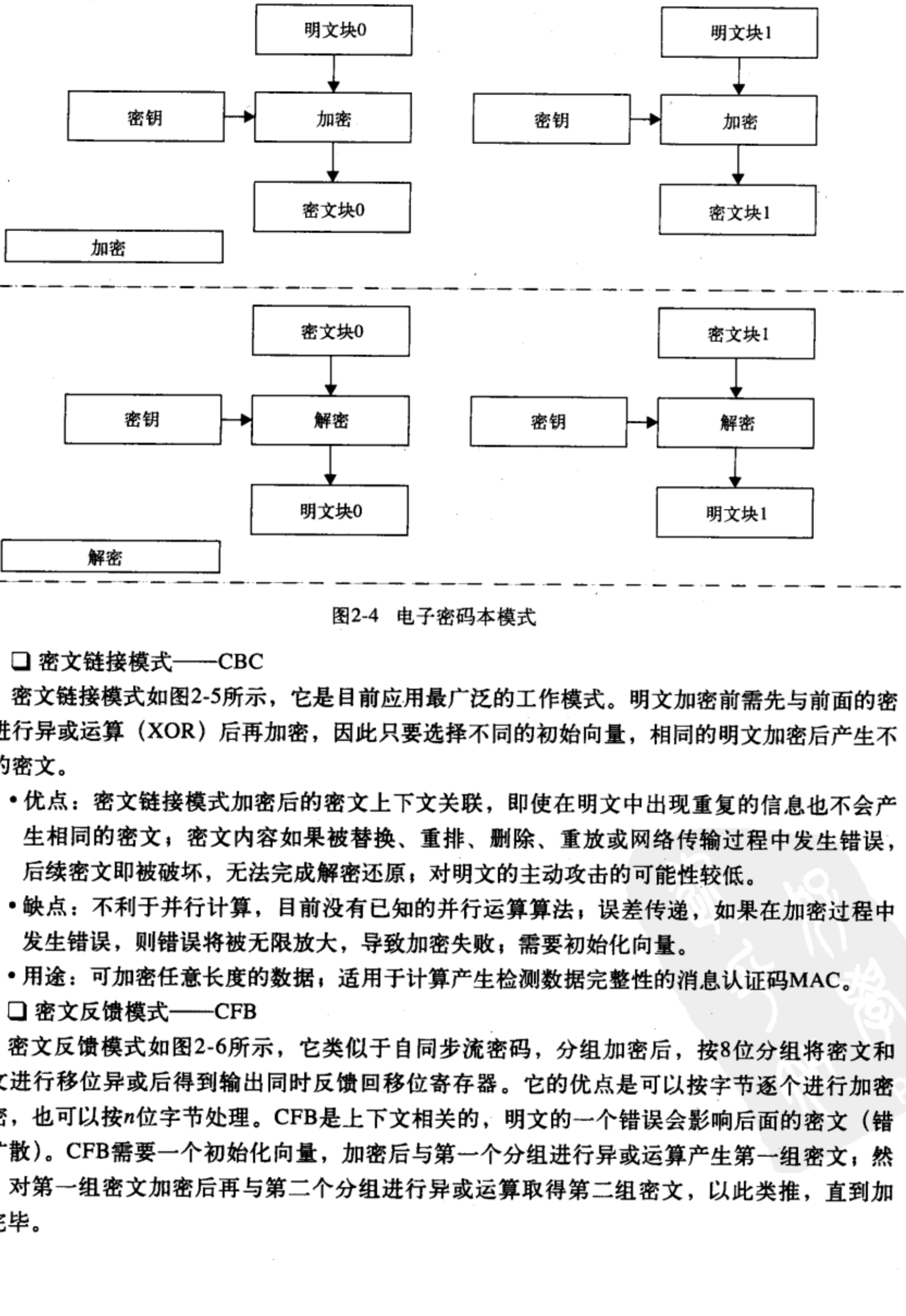


图2-4 电子密码本模式

❑ 密文链接模式——CBC

密文链接模式如图2-5所示，它是目前应用最广泛的工作模式。明文加密前需先与前面的密文进行异或运算（XOR）后再加密，因此只要选择不同的初始向量，相同的明文加密后产生不同的密文。

- 优点：密文链接模式加密后的密文上下文关联，即使在明文中出现重复的信息也不会产生相同的密文；密文内容如果被替换、重排、删除、重放或网络传输过程中发生错误，后续密文即被破坏，无法完成解密还原；对明文的主动攻击的可能性较低。
- 缺点：不利于并行计算，目前没有已知的并行运算算法；误差传递，如果在加密过程中发生错误，则错误将被无限放大，导致加密失败；需要初始化向量。
- 用途：可加密任意长度的数据；适用于计算产生检测数据完整性的消息认证码MAC。

❑ 密文反馈模式——CFB

密文反馈模式如图2-6所示，它类似于自同步流密码，分组加密后，按8位分组将密文和明文进行移位异或后得到输出同时反馈回移位寄存器。它的优点是可以按字节逐个进行加密解密，也可以按*n*位字节处理。CFB是上下文相关的，明文的一个错误会影响后面的密文（错误扩散）。CFB需要一个初始化向量，加密后与第一个分组进行异或运算产生第一组密文；然后，对第一组密文加密后再与第二个分组进行异或运算取得第二组密文，以此类推，直到加密完毕。

更多资源请访问稀酷客(www.ckook.com)

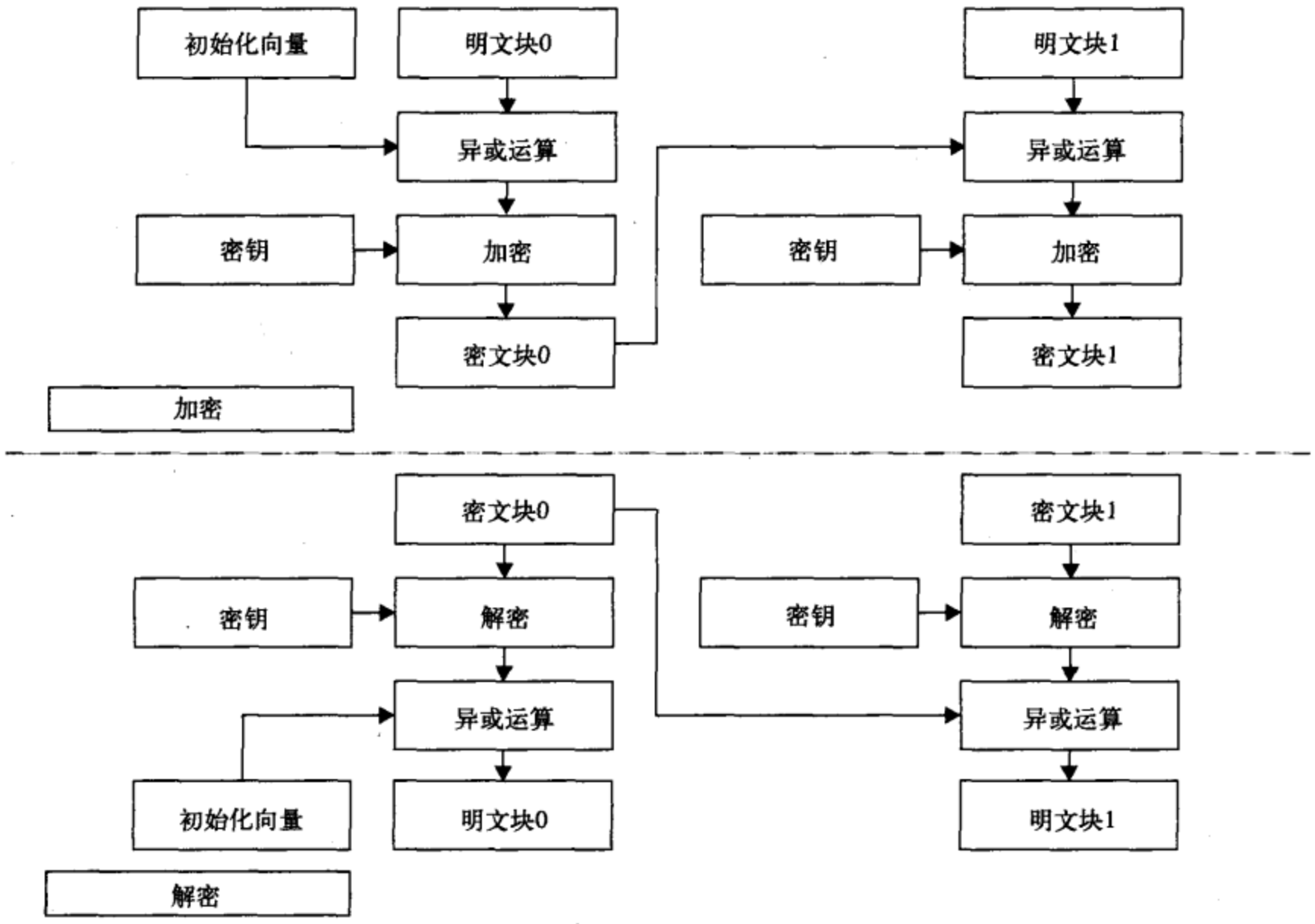


图2-5 密文链接模式

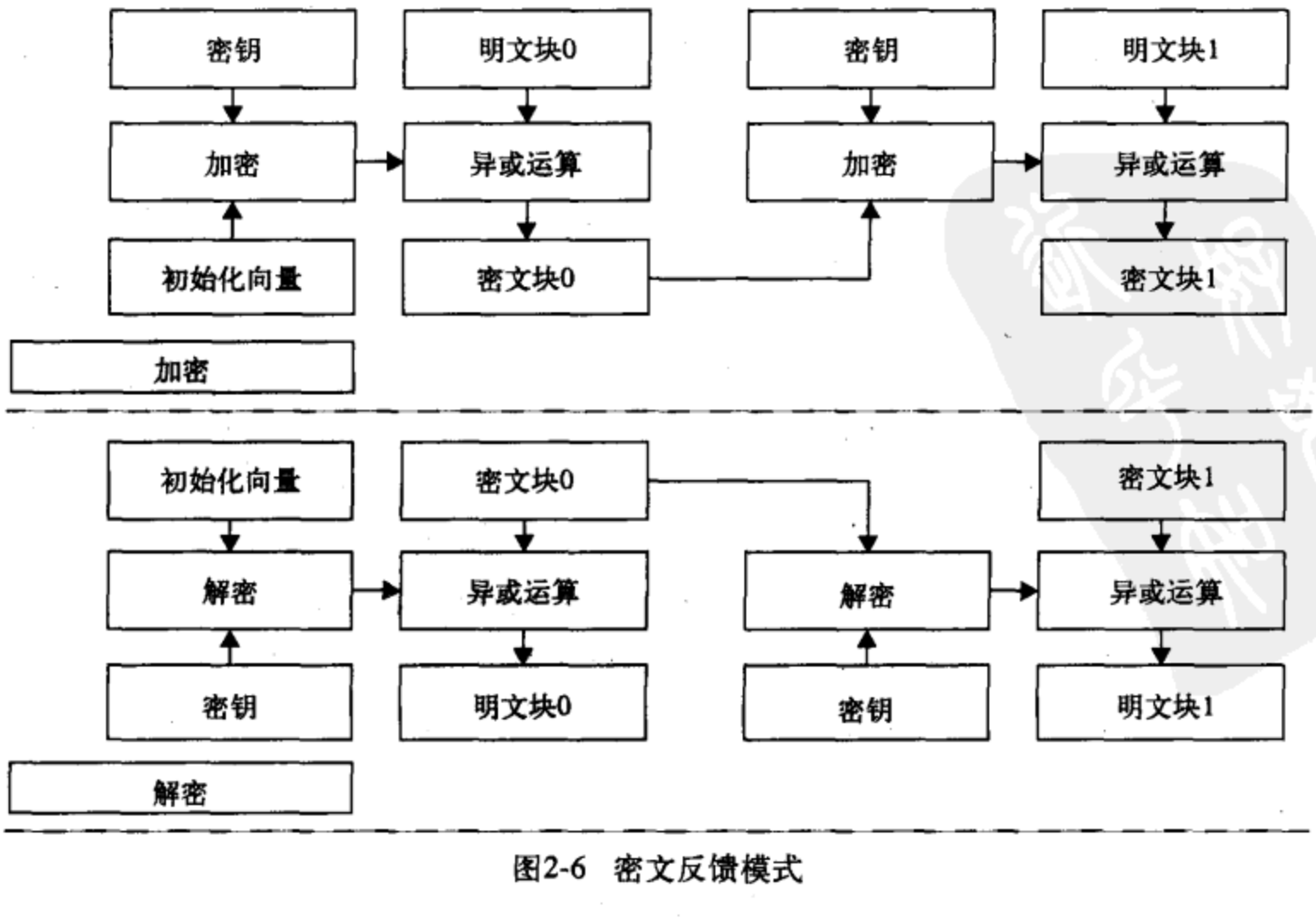


图2-6 密文反馈模式

更多资源请访问稀酷客(www.ckook.com)

- 优点：隐藏了明文的模式，每一个分组的加密结果必受其前面所有分组内容的影响，即使出现多次相同的明文，均产生不同的密文；分组密码转化为流模式，可产生密钥流，可以及时加密传送小于分组的数据。
- 缺点：与CBC相类似。不利于并行计算，目前没有已知的并行运算算法；存在误差传送，一个单元损坏影响多个单元；需要初始化向量。
- 用途：因错误传播无界，可用于检查发现明文密文的篡改。

❑ 输出反馈模式——OFB

输出反馈模式如图2-7所示，它将分组密码作为同步流密码运行，和CFB相似，不过OFB用的是前一个n位密文输出分组反馈回移位寄存器，OFB没有错误扩散问题。该模式产生与明文异或运算的密钥流，从而产生密文，这一点与CFB大致相同，唯一的差异是与明文分组进行异或的输入部分是反复加密后得到的。

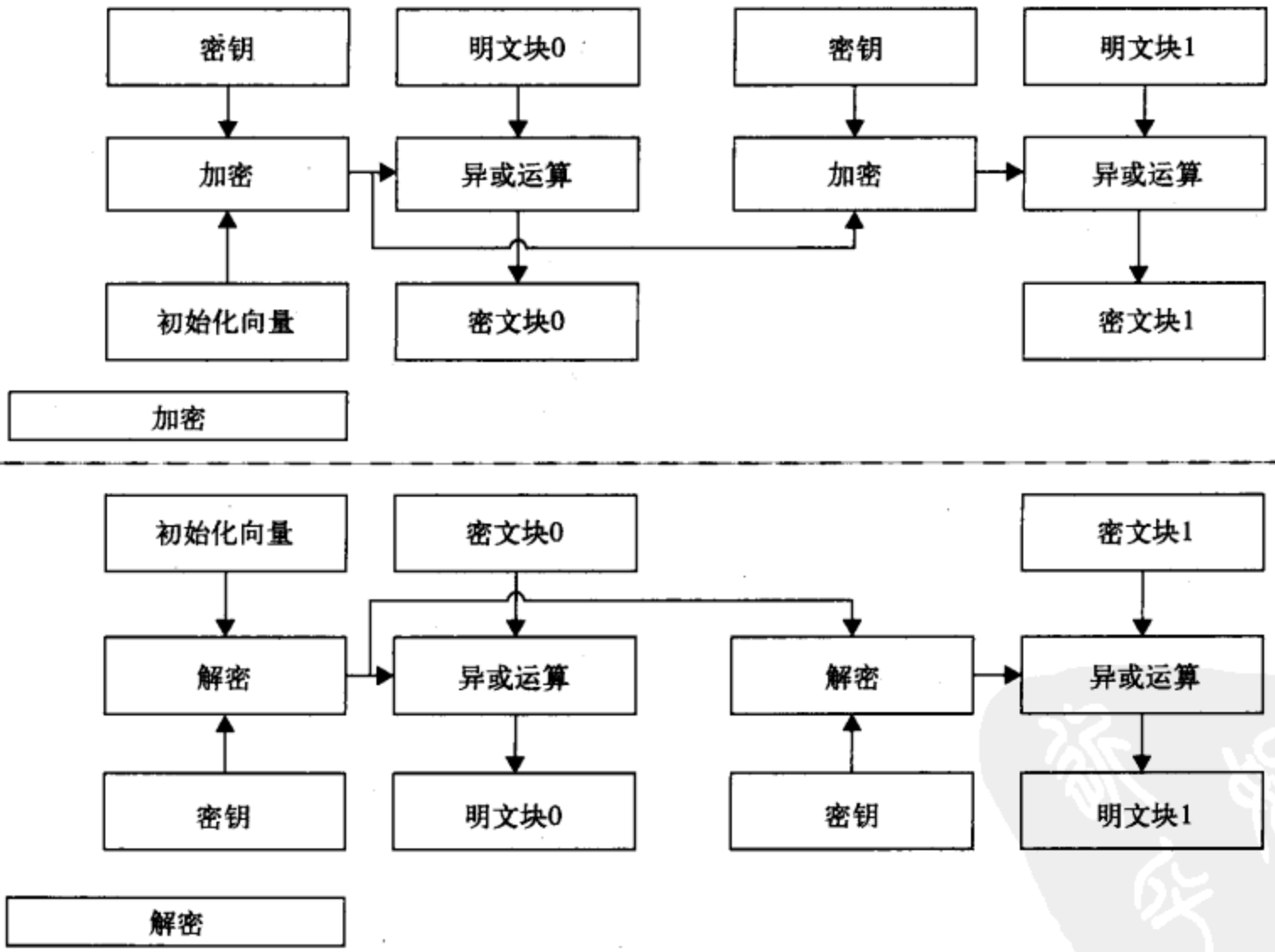


图2-7 输出反馈模式

- 优点：隐藏了明文的模式；分组密码转化为流模式；无误差传送问题；可以及时加密传送小于分组的数据。
- 缺点：不利于并行计算；对明文的主动攻击是可能的，安全性较CFB差。
- 用途：适用于加密冗余性较大的数据，比如语音和图像数据。

❑ 计数器模式——CTR

更多资源请访问稀酷客(www.ckook.com)

计数器模式如图2-8所示，它的特点是将计数器从初始值开始计数所得到的值发送给分组密码算法。随着计数器的增加，分组密码算法输出连续的分组来构成一个位串，该位串被用来与明文分组进行异或运算。计数器模式是用来提取分组密码的最大效能以实现保密性的。在AES的实际应用中，经常会选择CBC模式和CTR模式，但更多的是选择CTR模式。

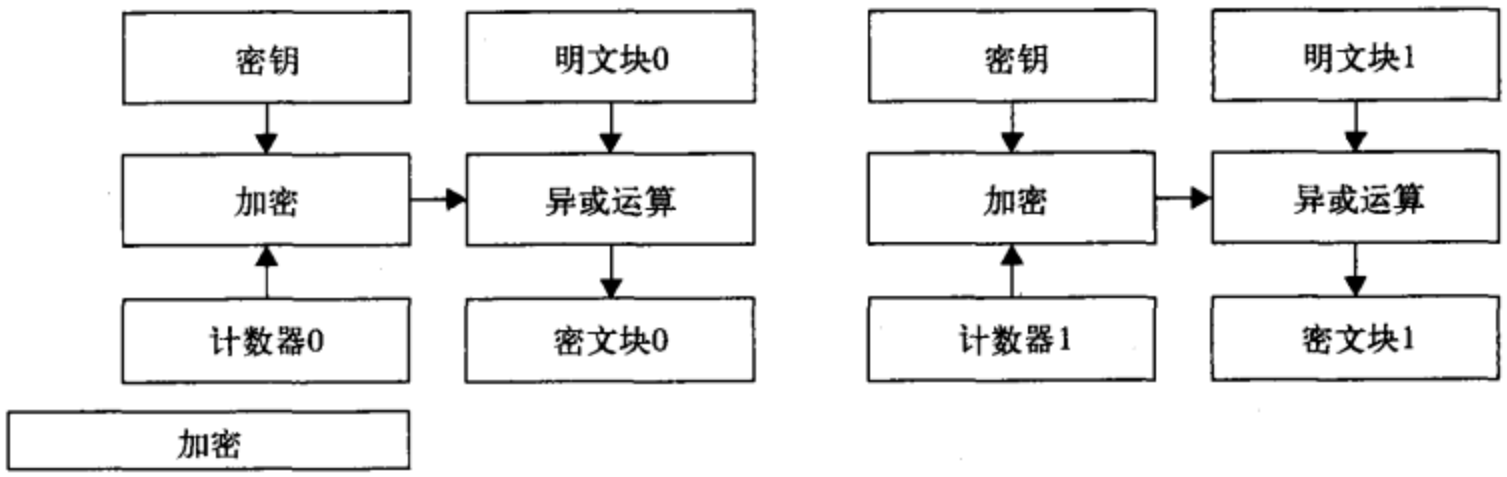


图2-8 计数器模式

- 优点：可并行计算，安全性至少与CBC模式一样好；加密与解密仅涉及密码算法的加密。
- 缺点：没有错误传播，因此不易确保数据完整性。
- 用途：适用于各种加密应用。

2.6 非对称密码体制

1976年，密码学专家Diffie和Hellman在《密码学的新方向》一文中提出了公开密钥密码体制的思想，开创了现代密码学的新领域，非对称密码体制的篇章由此揭开。

非对称密码体制的保密通信模型如图2-9所示。非对称密码体制与对称密码体制相对，其主要的区别在于：非对称密码体制的加密密钥和解密密钥不相同，分为两个密钥，一个公开，一个保密。公开的密钥称为公钥，保密的密钥称为私钥。因此，非对称密码体制也称为公钥密码体制。非对称密码体制使得发送者和接收者无密钥传输的保密通信成为可能，弥补了对称密码体制的缺陷。

更多资源请访问稀酷客(www.ckook.com)

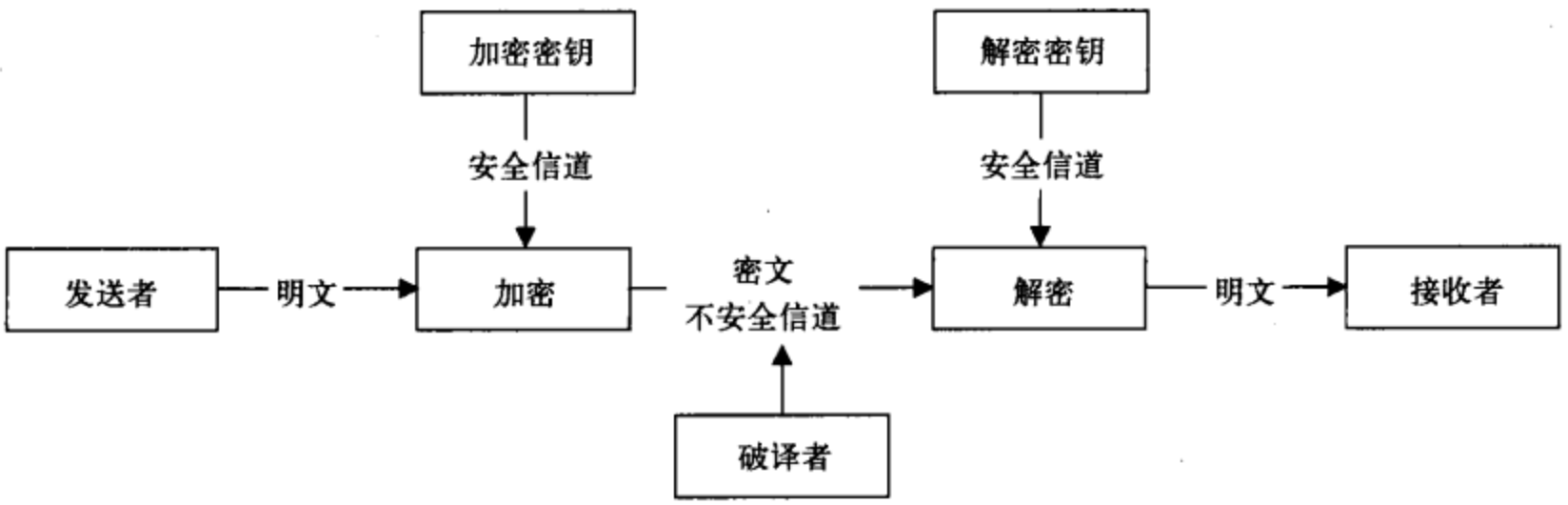


图2-9 非对称密码体制的保密通信模型

在非对称密码体制中，公钥和私钥均可用于加密与解密操作，但它与对称密码体制有极大的不同。公钥与私钥分属通信双方，一份消息的加密与解密需要公钥与私钥共同参与。公钥加密则需要私钥解密，反之，私钥加密则需要公钥解密。我们把通信双方定义为甲乙两方，甲乙两方分场景扮演信息发送者或接收者。公钥与私钥分属甲乙两方，甲方拥有私钥，乙方拥有公钥。为了更好地描述非对称密码体制通信流程，我们通过图2-10、图2-11来说明甲乙双方如何完成一次完整的会话。

甲方（发送方）用私钥加密数据向乙方发送数据，乙方（接收方）接收到数据后使用公钥解密数据，如图2-10所示。

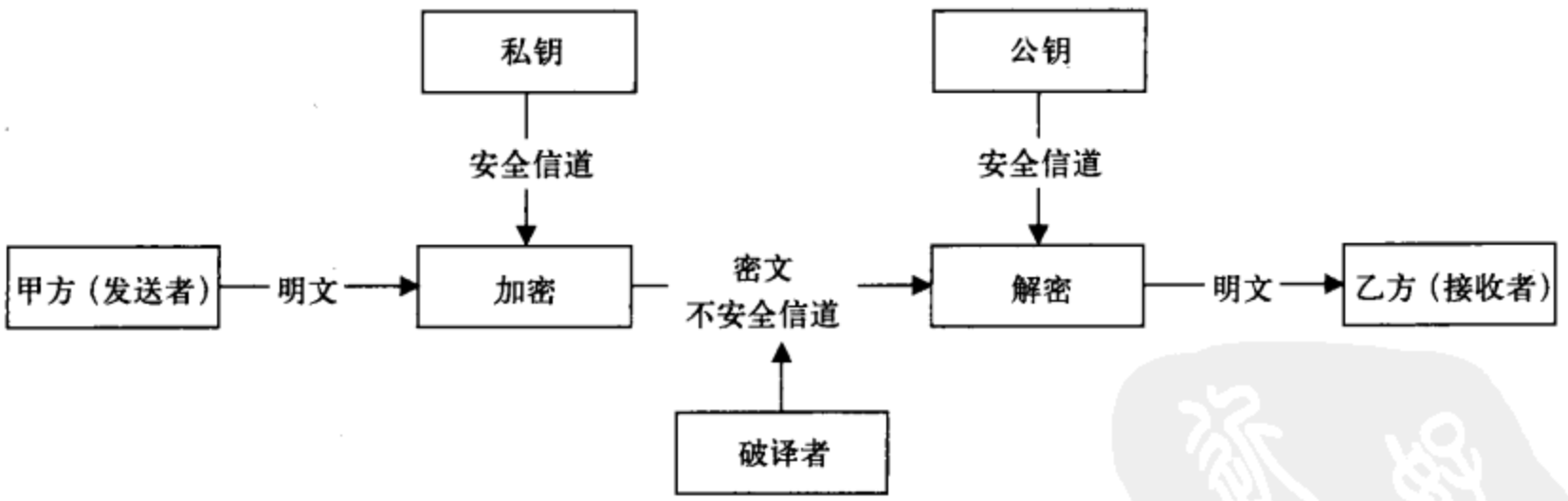


图2-10 私钥加密-公钥解密的保密通信模型

乙方（发送方）用公钥加密数据向甲方发送数据，甲方（接收方）接收到数据后使用私钥解密数据，如图2-11所示。

非对称密码体制的主要优点是可以适应开放性的使用环境，密钥管理问题相对简单，可以方便、安全地实现数字签名和验证。RSA是非对称密码体制的典范，它不仅可以完成一般的数据保密操作，同时它也支持数字签名与验证。关于数字签名，请参见2.8节。除了数字签名，非对称密码体制还支持数字信封等技术。我们将在后续章节详细讲述该类技术的具体实现。

非对称密码算法的安全性完全依赖于基于计算复杂度上的难题，通常来自于数论。例如，RSA源于整数因子分解问题；DSA——数字签名算法，源于离散对数问题；ECC——椭圆曲线

更多资源请访问稀酷客(www.ckook.com)

加密算法，源于离散对数问题。由于这些数学难题的实现多涉及底层模数乘法或指数运算，相对于分组密码需要更多的计算资源。为了弥补这一缺陷，非对称密码系统通常是复合式的：用高效率的对称密码算法对信息进行加密解密处理；用非对称密钥加密对称密码系统所使用的密钥。通过这种复合方式增进效率。

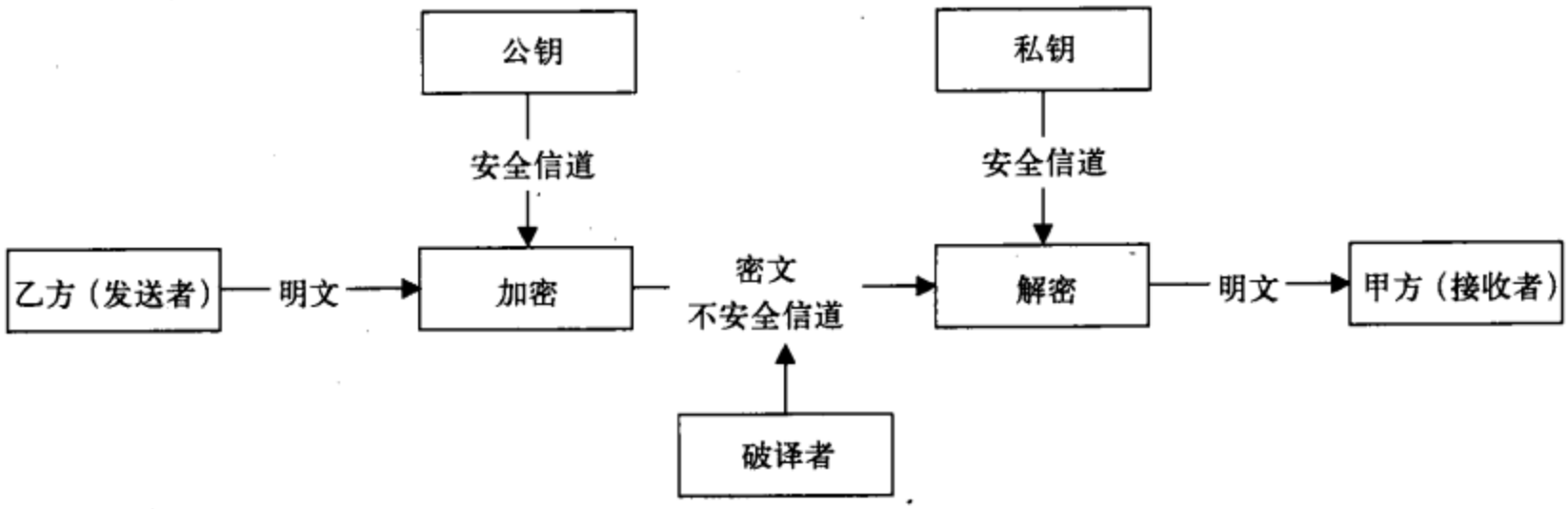


图2-11 公钥加密-私钥解密的保密通信模型

2.7 散列函数

在讲到对称密码体制的流密码实现方式时，曾经提到过对于信息完整性验证需要其他技术来支持，这种技术就是由散列函数提供的消息认证技术。

散列函数，也称做哈希函数、消息摘要函数、单向函数或杂凑函数。与上述密码体制不同的是，散列函数的主要作用不是完成数据加密与解密的工作，它是用来验证数据的完整性的重要技术。通过散列函数，可以为数据创建“数字指纹”（散列值）。散列值通常是一个短的随机字母和数字组成的字符串。消息认证流程如图2-12所示。

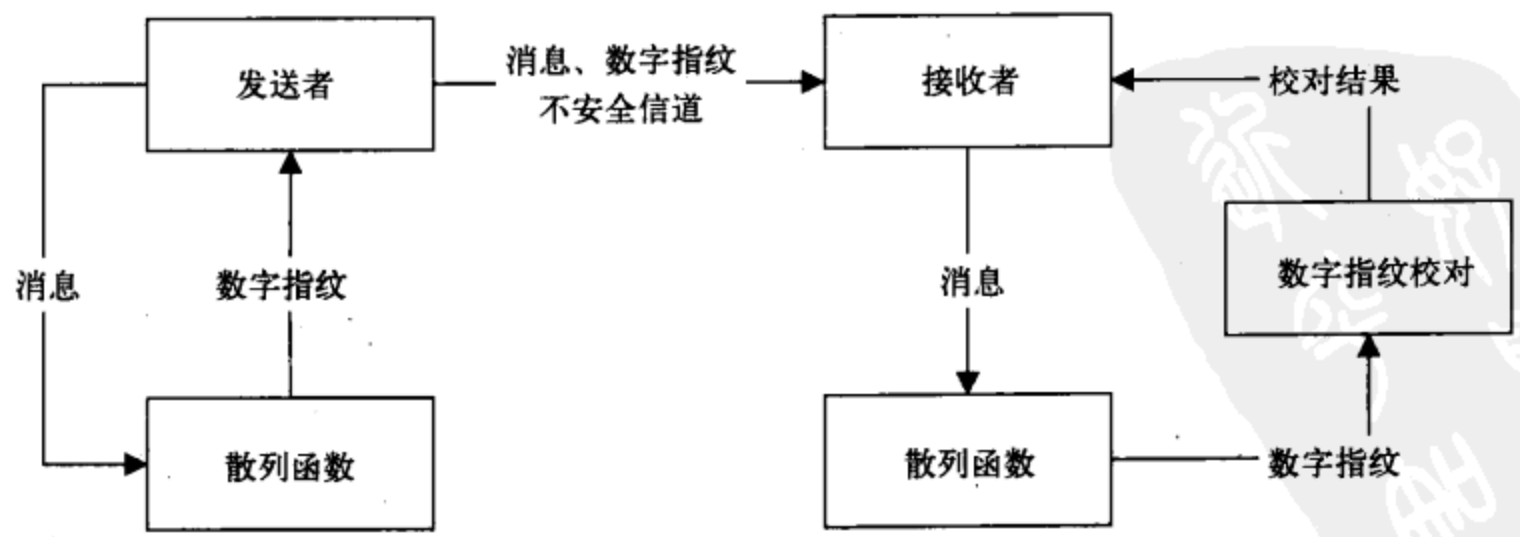


图2-12 消息认证流程

在上述认证流程中，信息收发双方在通信前已经商定了具体的散列算法，并且该算法是公开的。如果消息在传递过程中被篡改，则该消息不能与已获得的数字指纹相匹配。

散列函数具有以下一些特性：

更多资源请访问稀酷客(www.ckook.com)

- 消息的长度不受限制。
- 对于给定的消息，其散列值的计算是很容易的。
- 如果两个散列值不相同，则这两个散列值的原始输入消息也不相同，这个特性使得散列函数具有确定性的结果。
- 散列函数的运算过程是不可逆的，这个特性称为函数的单向性。这也是单向函数命名的由来。
- 对于一个已知的消息及其散列值，要找到另一个消息使其获得相同的散列值是不可能的，这个特性称为抗弱碰撞性。这被用来防止伪造。
- 任意两个不同的消息的散列值一定不同，这个特性称为抗强碰撞性。

散列函数广泛用于信息完整性的验证，是数据签名的核心技术。散列函数的常用算法有MD——消息摘要算法、SHA——安全散列算法及MAC——消息认证码算法。我们将在后续章节详述上述散列函数的算法实现。

2.8 数字签名

通过散列函数可以确保数据内容的完整性，但这还远远不够。此外，还需要确保数据来源的可认证（鉴别）性和数据发送行为的不可否认性。完整性、认证性和不可否认性，正是数字签名的主要特征。数字签名针对以数字形式存储的消息进行处理，产生一种带有操作者身份信息的编码。执行数字签名的实体称为签名者，签名过程中所使用的算法称为签名算法（Signature Algorithm），签名操作中生成的编码称为签名者对该消息的数字签名。发送者通过网络将消息连同其数字签名一起发送给接收者。接收者在得到该消息及其数字签名后，可以通过一个算法来验证签名的真伪以及识别相应的签名者。这一过程称为验证过程，其过程中使用的算法称为验证算法（Verification Algorithm），执行验证的实体称为验证者。数字签名离不开非对称密码体制，签名算法受私钥控制，且由签名者保密；验证算法受公钥控制，且对外公开。RSA算法则既是最为常用的非对称加密算法，也是最为常用的签名算法。DSA算法是典型的数字签名算法，虽然本身属于非对称加密算法不具备数据加密与解密的功能。

数字签名满足以下三个基本要求：

- 签名者任何时候多无法否认自己曾经签发的数字签名。
- 信息接收者能够验证和确认收到的数字签名，但任何人无法伪造信息发送者的数字签名。
- 当收发双方对数字签名的真伪产生争议时，通过仲裁机构（可信赖的第三方）进行仲裁。

数字签名认证流程如图2-13所示。在这里提请大家注意：私钥用于签名，公钥用于验证。签名操作只能由私钥完成，验证操作只能有公钥完成；公钥与私钥成对出现，用公钥加密的消息只能用私钥解密，用私钥加密的消息只能用公钥解密。

那么，数字签名认证是怎样一个流程呢？我们暂定甲方拥有私钥，并且甲方将公钥发布给乙方；当甲方作为消息的发送方时，甲方使用私钥对消息做签名处理，然后将消息加密后连同数字签名发送给乙方。乙方使用已获得的公钥对接收到的加密消息做解密处理，然后使用公钥

更多资源请访问稀酷客(www.ckook.com)

及数字签名对原始消息做验证处理。

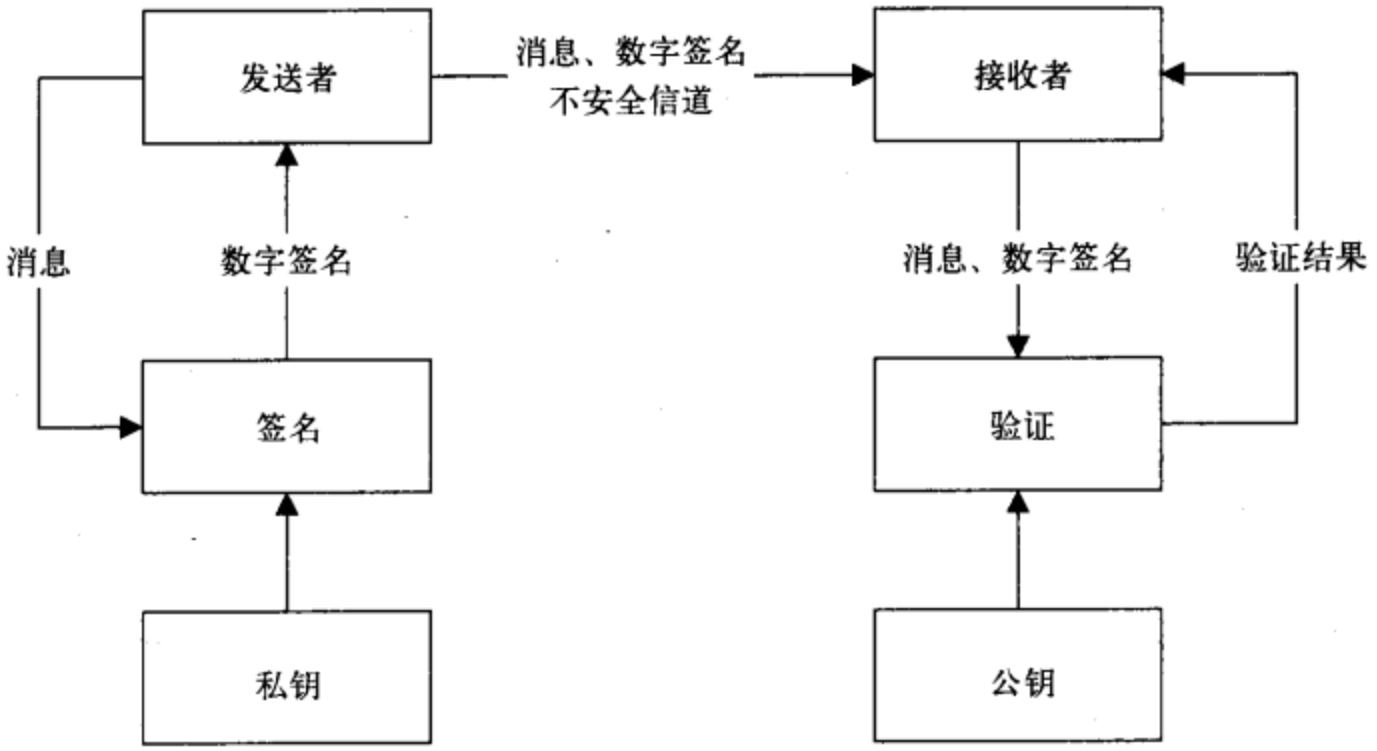


图2-13 数字签名认证流程

当然，我们可以对消息先加密，然后对加密后的消息做签名处理，这样乙方获得消息后，先做验证处理，如果验证通过则对消息解密。反之，验证失败则抛弃消息。这样做显然可以提高系统的处理速度，但即便如此，作者仍建议大家对消息先做签名，再做加密处理。加密与签名都应该只针对原始消息（明文）做处理。加密是为了确保消息在传送过程中避免被破解，签名是为了确保消息的有效性。消息本身可能就是一个可执行的文件，消息的接收方通过对消息的验证来判别该文件是否有权执行，而这个文件本身是不需要加密的。

由于签名不可伪造，甲方不能否认自己已发送的消息，而乙方可验证消息的来源以及消息是否完整。数字签名可提供OSI参考模型五类安全服务中的三种服务：认证（鉴别）服务、抗否认服务和数据完整性服务。正因如此，数字签名成为公钥基础设施以及许多网络安全机制的基础。

在上述认证过程的描述中，似乎大家有这样一个疑问：当乙方作为发送方，通过公钥将消息加密后发送给甲方时，由于算法、公钥公开，任何一个已获得公钥的窃听者都可以截获乙方发送的消息，替换成自己的消息发送给甲方，而甲方无法辨别消息是否来源于乙方。也就是说，上述的认证方式是单向的，属于单向认证。如果有两套公私钥，甲乙两方都对数据做签名及验证就可以避免这一问题。没错，这种认证方式正是双向认证。以网银交易为例，一般的网银交易使用的都是单向认证方式，无法验证使用者的身份；而要求较高的网银交易则都是双向认证方式，交易双方身份都可以得到验证。

2.9 密码学的未来

密码学历经四千年锤炼，从古代走到了现代，从军用走向了民用，逐步贴近我们生活领域

更多资源请访问稀酷客(www.ckook.com)

的每一个角落。密码学也有“新陈代谢”，各种国际著名的密码算法被攻破，预示着更加安全的密码算法即将诞生。

2.9.1 密码算法的破解

密码算法并不像我们想象的那么安全，我们所熟知的、常用的各种算法竟然在历史的昨天就被破解。

1997年1月28日，美国的RSA数据安全公司在RSA安全年会上公布了一项“秘密密钥挑战”竞赛，其中包括悬赏1万美元破译密钥长度为56位的DES。位于美国科罗拉多州的程序员Verser从1997年2月18日起，用了96天时间，在网络上数万名志愿者的协同工作下，成功地找到了DES的密钥，赢得了悬赏的1万美元。

1998年7月，电子前沿基金会（EFF）使用一台25万美元的电脑在56小时内破译了密钥长度为56位的DES。

1999年1月，RSA数据安全会议期间，电子前沿基金会用22小时15分钟就宣告破解了一个DES的密钥。

2004年8月，在美国加州圣芭芭拉召开的国际密码大会（Crypto'2004）上，山东大学王小云教授宣告她和她的团队已经破解了MD5、HAVAL-128、MD4和RIPEMD四大国际著名密码算法。MD5算法的破解预示着SHA-1算法的末日。

2005年2月，较之MD5算法有着更高安全系数的SHA-1算法毫无悬念地被王小云教授破解了。MD5和SHA-1的破解，动摇了目前数字签名的理论根基，从理论上说明数字签名可以伪造。

2007年，Marc Stevens、Arjen K. Lenstra和Benne de Weger进一步指出通过伪造软件签名，可重复性攻击MD5算法。研究者使用前缀碰撞法（chosen-prefix collision），使程序前端包含恶意程序，利用后面的空间添上垃圾代码凑出同样的MD5散列值。

2008年，荷兰埃因霍芬技术大学科学家成功把两个可执行文件进行了MD5碰撞，使得这两个运行结果不同的程序被计算出同一个MD5。2008年12月一组科研人员通过MD5碰撞成功生成了伪造的SSL证书，这使得在HTTPS协议中服务器可以伪造一些根CA的签名。

尽管各种大名鼎鼎的密码算法被破解，但这却更加带动了密码学前进的脚步。也许不久的将来，量子密码学将成为密码学领域新一代的霸主！

2.9.2 密码学的明天

随着计算机网络应用的迅猛发展，人们对信息安全和保密的重要性认识不断提高，密码学在信息安全中起着举足轻重的作用，已成为信息安全中不可或缺的重要部分。从古代发展到现代，由军用转为民用，密码学有着广泛的发展前景。自动柜员机芯片卡、公交IC卡、电子商务等都离不开密码学的支持。几乎可以说有网络的地方，就有密码学的身影。随着各种具有高度安全系数的国际密码算法的破解，密码算法正经历着自己的“新陈代谢”。新的密码学理论层出不穷，新的密码算法崭露头角。我们坚信，密码学的明天将有无限可能。

更多资源请访问稀酷客(www.ckook.com)

2.10 小结

纵观密码学的发展史，它的发展共经历了三个发展阶段，分别是手工加密阶段、机械加密阶段和计算机加密阶段。手工加密阶段最为漫长，期间孕育了古典密码，为后期密码学的发展奠定了基础。机械工业革命发展的同时促进着各种科学技术的进步，密码学也不例外。加之两次世界大战，更加促进了密码学的飞速发展，密码学由此进入现代密码学阶段。但尽管如此，在这一阶段的密码学仍旧未能摆脱古典密码学的影子，加密与解密操作均有赖于语言学的支持，转轮密码机Enigma的发明与破解更是将这一特点发挥到了极致。随着数据理论逐步介入，密码学逐渐成为一门学科，而非一门艺术。进入计算机加密阶段后，密码学应用不再局限于军事、政治和外交领域，逐步扩大到商务、金融和社会的其他领域。密码学的研究和应用已大规模扩展到了民用方面。

密码学主要包含两个分支：密码编码学和密码分析学。密码编码学针对于信息如何隐藏；密码分析学针对于信息如何破译。编码学与分析学相互影响，共同促进密码学的发展。

古典密码是现代密码的基础，移位和替代是古典密码最常用、最核心的两种加密技巧。由此，古典密码主要分为移位密码和替代密码。例如，凯撒密码就是替代密码的典范。替代密码其分支众多，包含单表替代密码、同音替代密码、多表替代密码和多字母替代密码。移位和替代技巧仍是现代密码学最常用的两种加密手段。

基于柯克霍夫原则，对密码算法公开，对密钥保密。密码算法公开有助于提高算法的安全性，避免算法自身的漏洞，如算法的设计者为算法留有后门等。

从密码体制上划分，现代密码学共分为两种密码体制：对称密码体制和非对称密码体制。对称与非对称的差别源于加密密钥和解密密钥是否对称，即加密密钥与解密密钥是否相同(对称)。

在对称密码体制中，加密与解密操作使用相同的密钥，我们把这个密钥称为秘密密钥。DES、AES算法都是常用的对称密码算法。流密码和分组密码都属于对称密码体制。流密码实现简单，对环境要求低，适用于手机平台的加密，广泛应用于军事、外交领域。RC4算法就是典型的流密码算法。流密码的理论、算法受限于国家安全因素未能公布。分组密码在这一点上与流密码恰恰相反，其理论、算法公开，分类众多。DES、AES算法等主要的对称密码算法均属于分组密码。分组密码共有五种工作模式：电子密码本模式（ECB）、密文链接模式（CBC）、密文反馈模式（CFB）、输出反馈模式（OFB）、计数器模式（CTR）。分组密码会产生短块，关于短块的处理方法有填充法、流密码加密法、密文挪用技术。

在非对称密码体制中，加密与解密操作使用不同的密钥。对外公开的密钥，称为公钥；对外保密的密钥，称为私钥。用公钥加密的数据，只能用私钥解密；反之，用私钥加密的数据，只能用公钥解密。RSA算法是常用的非对称密码算法。非对称密码体制同时支持数字签名技术，如RSA、DSA都是常用的数字签名算法。

散列函数可以有效地确保数据完整性，被作为消息认证技术。常用的散列函数算法有MD5、SHA、MAC。散列函数也是数字签名技术中最重要的技术环节。数字签名离不开非对称密码

更多资源请访问稀酷客(www.ckook.com)

体制，其私钥用于签名，公钥用于验证。基于数字签名的不可伪造性，数字签名技术成为五类安全服务中数据完整性服务、认证性服务和抗否认性服务的核心技术。通信双方只有一方提供数字签名的认证方式称为单向认证，通信双方都提供数字签名的认证方式称为双向认证。一般网银系统多采用单向认证方式，而要求较高的网银交易则都采用双向认证方式。密码学在不断地向前发展，只不过它的发展通常是以其密码算法的破解而引发，以更高安全系数算法的诞生而告一段落，密码学的明天将无可限量。



更多资源请访问稀酷客(www.ckook.com)

第3章

Java加密利器

通过前第1、2章内容的学习，我们已经对密码学的理论有了一定的认识。那么，该如何将如此深奥却又如此关键的技术运用到我们的应用平台中呢？先别急，“工欲善其事，必先利其器”，先看看我们手里都有哪些工具可以驾驭这枚银弹！

从本章开始，我们将进入本书的主题——Java加密与解密的艺术。

3.1 Java与密码学

离开了安全问题，密码学就没有存在的价值。因此，在Java的世界里，密码学是其安全模块的重要组成部分。

3.1.1 Java安全领域组成部分

Java安全领域总共分为四个部分：JCA（Java Cryptography Architecture，Java加密体系结构）、JCE（Java Cryptography Extension，Java加密扩展包）、JSSE（Java Secure Sockets Extension，Java安全套接字扩展包）、JAAS（Java Authentication and Authentication Service，Java鉴别与安全服务）。

- JCA提供基本的加密框架，如证书、数字签名、消息摘要和密钥对产生器。
- JCE在JCA的基础上作了扩展，提供了各种加密算法、消息摘要算法和密钥管理等功能。我们已经有所了解的DES算法、AES算法、RSA算法、DSA算法等就是通过JCE来提供的。有关JCE的实现主要在javax.crypto包（及其子包）中。
- JSSE提供了基于SSL（Secure Sockets Layer，安全套接字层）的加密功能。在网络的传输过程中，信息会经过多个主机（很有可能其中一台就被窃听），最终传送给接收者，这是不安全的。这种确保网络通信安全的服务就是由JSSE来提供的。
- JAAS提供了在Java平台上进行用户身份鉴别的功能。如何提供一个符合标准安全机制的登录模块，通过可配置的方式集成至各个系统中呢？这是由JAAS来提供的。

本书将要讨论的主要是JCA、JCE和JSSE。

JCA和JCE是Java平台提供的用于安全和加密服务的两组API。它们并不执行任何算法，

更多资源请访问稀酷客(www.ckook.com)

它们只是连接应用和实际算法实现程序的一组接口。软件开发商可以根据JCE接口（也叫安全提供者接口）将各种算法实现后，打包成一个Provider（安全提供者），动态地加到Java运行环境中。

根据美国出口限制规定，JCA可出口（JCA和Sun的一些默认实现包含在Java发行版中），但JCE对部分国家是限制出口的。因此，要实现一个完整的安全结构，就需要一个或多个第三方厂商提供的JCE产品，称为安全提供者。BouncyCastle JCE就是其中的一个安全提供者。

安全提供者是承担特定安全机制的实现的第三方。有些提供者是完全免费的，而另一些提供者则需要付费。提供安全提供者的公司有Sun、Bouncy Castle等，Sun提供了如何开发安全提供者的细节。Bouncy Castle提供了可以在J2ME/J2EE/J2SE平台得到支持的API，而且Bouncy Castle的API是免费的。

JDK 1.4版本及其后续版本中包含了上述扩展包，无须进行相应配置。在此之前，安装JDK后需要对上述扩展包进行相应配置。

Java安全体系结构通过扩展的方式，加入了更多的算法实现及相应的安全机制。我们把这些提供者称为安全提供者（以下简称“提供者”）。下述Properties文件中列举了JDK 1.6版本所提供的安全提供者的配置信息：

```
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=com.sun.net.ssl.internal.ssl.Provider
security.provider.4=com.sun.crypto.provider.SunJCE
security.provider.5=sun.security.jgss.SunProvider
security.provider.6=com.sun.security.sasl.Provider
security.provider.7=org.jcp.xml.dsig.internal.dom.XMLDSigRI
security.provider.8=sun.security.smartcardio.SunPCSC
security.provider.9=sun.security.mscapi.SunMSCAPI
```

上述这些提供者均是Provider类（java. security.Provider）的子类。其中sun.security.provider.Sun是基本安全提供者，sun.security.rsa.SunRsaSign是实现RSA算法的提供者。

Java安全体系不仅支持来自Sun官方提供的安全提供者，同时也可配置第三方安全提供者以扩展相应的算法实现等。

安全提供者实现了两个概念的抽象：引擎和算法。引擎可以理解为操作，如加密、解密等。算法则定义了操作如何执行，如一个算法可以理解为一个引擎的具体实现。当然，一个算法可以有多种实现方式，这就意味着同一个算法可能与多个引擎的具体实现相对应。

安全提供者接口的目的就是提供一个简单的机制，从而可以很方便地改变或替换算法及其实现。在实际开发中，程序员只需要用引擎类实现特定的操作，而不需要关心实际进行运算的类是哪一个。

Provider类和Security类（java. security.Security）共同构成了安全提供者的概念。我们将在本章介绍这两个类的Java API。

更多资源请访问稀酷客(www.ckook.com)

3.1.2 关于出口的限制

密码学作为第二次世界大战时决胜关键的一项秘密武器，自然受到各个国家的重视。尤其是军事领域，其出口受到严格的限制。例如，美国以及其他国家都限制加密软件的出口与进口（Sun公司的产品是不能出口到缅甸的）。近年来，美国放松了加密软件出口的条件。尽管如此，JCE和JSSE出口仍受到种种限制。这种出口与进口的限制主要体现在JCE和JSSE的加密算法中。

虽然，核心Java API提供了相应的加密与解密实现，但其加密强度仍不具备军事意义。我们知道密钥长度通常与加密强度成正比，而我们所使用的由Sun公司提供的Java API，因受到美国军事出口限制在其密钥长度上有所缩减，加密强度有所降低。例如，DES算法因受到军事出口限制，目前仅提供56位的密钥长度，而事实上，安全要求则至少需要128位。除此之外，在算法实现上受军事出口限制，其实现内容本身保密，且分为军事与非军事两种实现。显然，非军事加密算法实现远远不如军事加密算法的加密强度高。

3.1.3 本书所使用的软件

本书将使用到Java软件包、测试工具JUnit软件包和Java开发工具Eclipse软件包三大部分。

1. 关于Java软件包

本书内容主要基于Java SE（1.6版本），本书中统称Java 6。Java SE包含两个部分：JDK和JRE，JRE通常包含在JDK中。

在本书中，我们将使用环境变量%JDK_HOME%来表示JDK的安装路径；使用环境变量%JRE_HOME%来表示JRE的安装路径。

如将JDK安装在C:\java\jdk目录下，变量%JDK_HOME%则指向该目录。相应地，如将JRE安装在C:\ java \jre目录下，变量%JRE_HOME%则指向该目录。

读者可通过Java官方网站（http://java.sun.com/javase/downloads/index.jsp）下载相应版本的JDK软件包。

在本书后面的描述中，如不做特殊说明均指Java 6版本。

2. 关于Java开发工具Eclipse 软件包

本书将使用常用的Java开发工具Eclipse来完成Java编译操作，其版本为3.5，代号Galileo（伽利略），见图3-1。读者可通过Eclipse官方网站（http://www.eclipse.org/downloads/）下载该软件包。

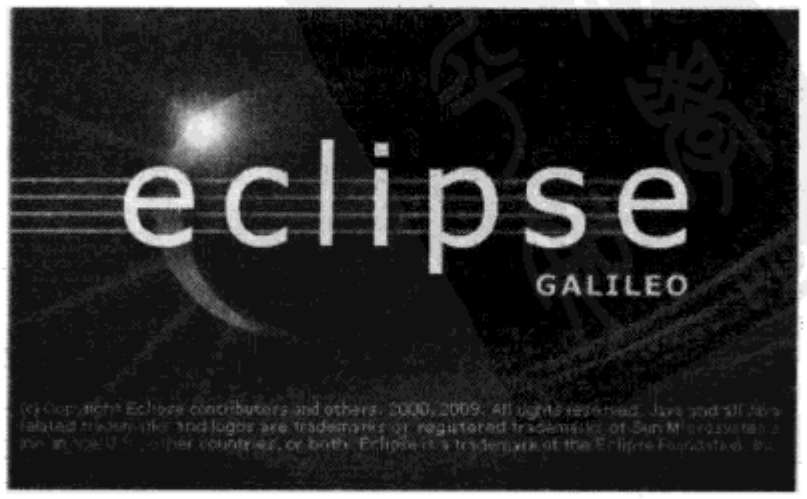


图3-1 Eclipse Galileo

相信大家对于如何使用Eclipse已经相当熟练了，因此，本书对于如何配置Eclipse不做详述。

3. 关于测试工具JUnit软件包

本书将通过测试工具JUnit，以白盒测试的方式演示如何使用Java完成相应的加密与解密

更多资源请访问稀酷客(www.ckook.com)

操作。

本书将使用JUnit 4.5版本，以注解的方式构建白盒测试。该测试框架已集成在本书所使用的Eclipse中。读者可通过其官方网站（http://www.junit.org/）下载最新的软件包。

4. 关于第三方开源组件包

- Bouncy Castle（http://www.bouncycastle.org/）是一个开源加密组件。它提供了多种Java 6所不支持的算法实现，如消息摘要算法MD4和SHA-224、对称加密算法IDEA、数字签名算法ECDSA等。本书中使用的版本为1.43。
- Commons Codec（http://commons.apache.org/codec/）同样是一款开源组件，它位于国际开源组织Apache（http://www.apache.org/）旗下。它对Java 6的API做了进一步封装，加强了易用性。本书中使用的版本为1.4。

本书将在第4章中详述以上两项内容。

5. 关于网络监听工具WireShark

本书将通过网络监听工具WireShark（如图3-2所示）完成对网络数据的监控，请读者参考相关文档并通过其官方网站（http://www.wireshark.org/）下载最新的软件包。

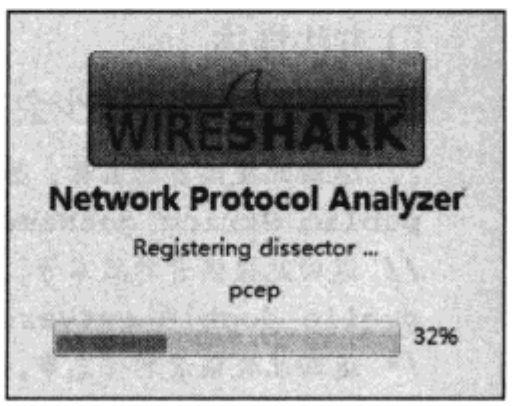


图3-2 WireShark

3.1.4 关于本章内容

本章将挑选与本书关联较为紧密的Java API作为主要内容。

本章主要详解了java.security包与javax.crypto包，这两个包中包含了Java加密与解密的核心部分，将是本书着重叙述的内容。

在java.security.interfaces包和javax.crypto.interfaces包中包含了密钥相关的接口，本章将在3.2节中简要介绍。

在java.security.spec包和javax.crypto.spec包中包含了密钥规范和算法参数规范的类和接口，本章将详述如何通过密钥规范获得相应的密钥。

3.1节将着重描述如何完成加密与解密的算法的实现，3.2节和3.3节将描述如何在Java开发环境中使用证书，以及如何通过HTTPS协议构建加密网络环境。

3.2 java.security包详解

java.security包为安全框架提供类和接口。对该包有所了解的读者一定知道，这只是探索Java加密世界的第一步。通过该包中的Java实现，仅仅能够完成消息摘要算法的实现（消息摘要处理的MessageDigest、DigestInputStream和DigestOutputStream类），并且其源代码是可见的。而要实现真正的加密与解密实现，需要参考javax.crypto包中的内容。当然，这个包中的源代码内容是不可见的。

对于该包中与本书相关程度较少的Java API内容（如java.security.Permission类，与JAAS相关）本书不做介绍。如果读者对这些内容有兴趣，请参考相应的Java API的内容。

更多资源请访问稀酷客(www.ckook.com)

3.2.1 Provider

Provider类实现了 Java 安全性的一部分或全部，我们称它为提供者，如下所示：

```
// 提供者抽象类
public abstract class Provider
extends Properties
```

Provider类可能实现的服务包括：

- 算法（如DSA、RSA、MD5或SHA-1）。
- 密钥的生成、转换和管理设施（如用于特定于算法的密钥）。

每个提供者都有一个名称和一个版本号，并且在它每次装入运行时中进行配置。

- 方法详述

在实际开发中，很少会直接使用Provider类，我们通常使用的是以下几种方法：

```
// 返回此提供者的名称，如 SunRsaSign 表示该提供者的名称。
public String getName()
// 返回此提供者的版本号，如1.5表示该提供者的版本号。
public double getVersion()
/* 返回此提供者的信息串。如Sun RSA signature provider表示这是一个
用于RSA算法的数字签名提供者。*/
public String getInfo()
```

Provider类覆盖了Object类的以下方法，用以输出自身提供者信息：

```
/* 返回包含此提供者的名称和版本号的字符串。如SunRsaSign version 1.5
表示提供者名称为SunRsaSign，其版本号为1.5*/
public String toString()
```

Provider类继承于Properties类，并覆盖了Properties类的几种常用方法。

我们先来看一下和线程安全相关的方法。

我们可以通过读取输入流的信息，载入提供者相关信息，如下所示：

```
// 从输入流中读取属性列表（键和元素对）。
public synchronized void load(InputStream inStream)
```

我们也可以通过下述方法添加提供者相关信息：

```
// 设置键属性，使其具有指定的值。
public synchronized Object put(Object key, Object value)
// 将指定 Map 中所有映射关系复制到此提供者中。
public synchronized void putAll(Map<?,?> t)
```

我们可以通过如下方法获得当前提供者的相关信息：

```
// 返回此提供者中所包含的属性项的一个不可修改的 Set 视图。
public synchronized Set<Map.Entry<Object,Object>> entrySet()
```

我们可以通过下述方法，清理提供者的相关信息：

```
// 移除键属性（及其相应的值）
public synchronized Object remove(Object key)
```

更多资源请访问稀酷客(www.ckook.com)

如果使用如下方法，将清理掉有关提供者所支持的全部算法信息：

```
// 清除此提供者，使其不再包含用来查找由该提供者实现的设施的属性。
public synchronized void clear()
```

接下来，我们来了解一下非线程安全相关的方法。

Provider类是Properties类的子类，提供了相应的键值操作方法。

以下两个方法均通过给定键获得相应的值：

```
// 返回指定键所映射到的值，如果此映射不包含此键的映射，则返回 null。
public Object get(Object key)
// 用指定的键在此属性列表中搜索属性。
public String getProperty(String key)
```

以下两个方法可以用枚举的方式获得相应的键值信息：

```
// 返回此散列表中的键的枚举。
public Enumeration<Object> keys()
// 返回此散列表中的值的枚举。
public Enumeration<Object> elements()
```

通过以下方法可以获得集合方式的键值信息：

```
// 返回此提供者中所包含的属性键的一个不可修改的 Set 视图。
public Set<Object> keySet()
// 返回此提供者中所包含的属性值的一个不可修改的 Collection 视图。
public Collection<Object> values()
```

Provider类覆盖上述Properties类方法的目的在于确保程序有足够的权限执行相应的操作。这方面涉及安全管理器的概念，已超出本书所讨论的范畴，有兴趣的读者可以了解SecurityManager类，它位于java.lang包内。

自Java 5开始，Provider类中加入了内部类——Service类。Service类封装了服务的属性，并提供一个用于获得该服务的实现实例的工厂方法。以下为Provider类对Service类的调用支持：

```
// 安全服务的描述。
public static class Provider.Service
// 获取描述此算法或别名的指定类型的此提供者实现的服务。
public synchronized Provider.Service getService(String type, String algorithm)
// 获取此提供者支持的所有服务的一个不可修改的 Set。
public synchronized Set<Provider.Service> getServices()
```

在这里讲述Provider类，目的在于引导大家了解“提供者”这一概念，故不对Service类做过多讲述。在Java的安全提供者体系结构中，安全提供者可能不是来自Sun本身。不同的算法可能需要由不同的提供者提供，甚至其提供者本身属于第三方。图3-3是作者所使用的Java开发环境下的Provider类及其子类。

相信大家看到位于sun.security.provider包中的Sun类时并不感到陌生。当大家看到位于org.bouncycastle.jce.provider包中的BouncyCastleProvider类时，一定会感到有些疑惑。从包名结构来看，这不像是由Sun提供的JCE引擎提供者。没错，这是由第三方开源组织Bouncy

更多资源请访问稀酷客(www.ckook.com)

Castle（http://www.bouncycastle.org/）来提供的。有关于Bouncy Castle加密组件配置及使用，请参考本书第4章相关内容。

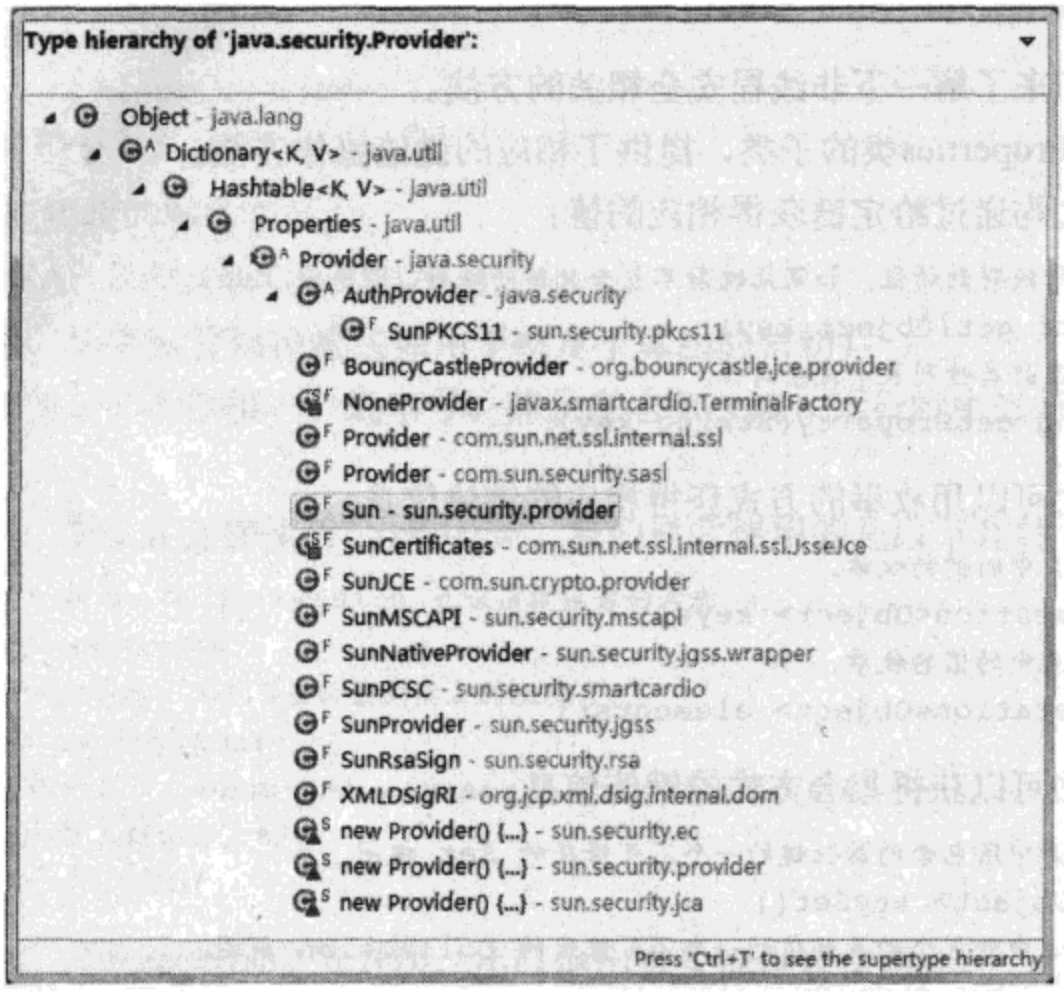


图3-3 Provier类及其子类

那么，在Java 6的环境中都有哪些提供者呢？

查看%JDK_HOME%\jre\lib\security目录，用记事本打开java.security文件。我们会发现这个Properties文件中可以找到以下9种安全提供者：

```
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=com.sun.net.ssl.internal.ssl.Provider
security.provider.4=com.sun.crypto.provider.SunJCE
security.provider.5=sun.security.jgss.SunProvider
security.provider.6=com.sun.security.sasl.Provider
security.provider.7=org.jcp.xml.dsig.internal.dom.XMLDSigRI
security.provider.8=sun.security.smartcardio.SunPCSC
security.provider.9=sun.security.mscapi.SunMSCAPI
```

上述这些配置是按照以下方式来配置的：

```
security.provider.<n>=<className>
```

很显然，为了加入Bouncy Castle加密组件的安全提供者只需要这样做：

```
#增加BouncyCastleProvider
security.provider.10=org.bouncycastle.jce.provider.BouncyCastleProvider
```

有关如何配置Bouncy Castle加密组件，请参考本书第4章相关内容。Provider类的常用方法

更多资源请访问稀酷客(www.ckook.com)

调用基本上是和Security类结合在一起的，有关Provider类方法的调用将在下一节中展示。

3.2.2 Security

Security类的任务就是管理Java 程序中所用到的提供者类。Security类是一个终态类，除了它的私有构造方法，其余均为静态方法。

```
// 管理提供者
public final class Security
extends Object
```

❑ 方法详述

Security类最主要的工作就是对提供者的管理。

我们可以通过如下方法向系统中追加一个新的提供者：

```
// 将提供者添加到下一个可用位置。在提供者数组尾部追加新的提供者。
public static int addProvider(Provider provider)
```

或者，我们直接指定这个提供者位于提供者列表中的位置，请看如下方法：

```
// 在指定的位置添加新的提供者。位置计数从1开始。
public static int insertProviderAt(Provider provider, int position)
```

这里有一个优先使用的问题，也就是说位置与优先级成正比，提供者位置越靠前，优先级越高。

能够向系统添加提供者，就能将其移除系统，可以使用如下方法：

```
// 移除带有指定名称的提供者。其余提供者在提供者数组中的位置将可能上移。
public static void removeProvider(String name)
```

当然，移除时需要提供提供者的缩写名称，如缩写BC指的是Bouncy Castle提供者。

与移除方法相类似，从系统中获得一个提供者可使用如下方法：

```
// 返回使用指定的名称安装的提供者（如果有）。
public static Provider getProvider(String name)
```

当然，也可以使用如下方法直接获得全部提供者：

```
// 返回包含所有已安装的提供者的数组（拷贝）。
public static Provider[] getProviders()
```

在获得全部提供者的前提下，我们也可以对提供者做一些过滤，可使用如下方法：

```
/* 返回包含满足指定的选择标准的所有已安装的提供者的数组（拷贝），如果尚未安装此类提供者，则返回 null。*/
public static Provider[] getProviders(Map<String,String> filter)
/* 返回包含满足指定的选择标准的所有已安装的提供者的数组（拷贝），如果尚未安装此类提供者，则返回 null。*/
public static Provider[] getProviders(String filter)
```

大家对%JDK_HOME%\jre\lib\security\java.security文件中的配置应该还有些印象吧？以下方法可以用来设置该文件的相关配置：

```
//设置安全属性值。
public static void setProperty(String key, String datum)
```

更多资源请访问稀酷客(www.ckook.com)

```
//获取安全属性值
public static String getProperty(String key)
```

要取到java.security文件中security.provider.1对应的值（sun.security.provider.Sun），或者对它进行设置就可以使用上述方法来实现。

除此之外，我们还可以通过下述方法获得指定加密服务所对应的可用算法或类型的名称：

```
/* 返回Set视图，这些Set视图中的字符串包含了指定的 Java 加密服务的所有可用算法或类型的名称（例如，Signature、MessageDigest、Cipher、Mac、KeyStore）。*/
public static Set<String> getAlgorithms(String serviceName)
```

❑ 实现示例

我们可以通过代码查看当前环境中的安全提供者信息，如代码清单3-1所示：

代码清单3-1 打印当前系统所配置的全部安全提供者

```
// 遍历目前环境中的安全提供者。
for (Provider p : Security.getProviders()) {
      // 打印当前提供者信息。
      System.out.println(p);
      // 遍历提供者Set实体。
      for (Map.Entry<Object, Object> entry : p.entrySet()) {
            // 打印提供者键值。
            System.out.println("\t" + entry.getKey());
      }
}
```

在上述程序执行后，我们将在控制台中看到以下内容：

```
SUN version 1.6
      Alg.Alias.Signature.SHA1/DSA
      Alg.Alias.Signature.1.2.840.10040.4.3
      Alg.Alias.Signature.DSS
      SecureRandom.SHA1PRNG ImplementedIn
      KeyStore.JKS
      Alg.Alias.MessageDigest.SHA-1
      MessageDigest.SHA
      KeyStore.CaseExactJKS
      CertStore.com.sun.security.IndexedCollection ImplementedIn
      Alg.Alias.Signature.DSA
      KeyFactory.DSA ImplementedIn
      KeyStore.JKS ImplementedIn
      AlgorithmParameters.DSA ImplementedIn
      Signature.NONEwithDSA
      Alg.Alias.CertificateFactory.X509
      CertStore.com.sun.security.IndexedCollection
      Provider.id className
      Alg.Alias.Signature.SHA-1/DSA
      CertificateFactory.X.509 ImplementedIn
      Signature.SHA1withDSA KeySize
```

更多资源请访问稀酷客(www.ckook.com)

```
KeyFactory.DSA
CertPathValidator.PKIX ImplementedIn
Configuration.JavaLoginConfig
Alg.Alias.Signature.OID.1.2.840.10040.4.3
Alg.Alias.KeyFactory.1.2.840.10040.4.1
MessageDigest.MD5 ImplementedIn
Alg.Alias.Signature.RawDSA
Provider.id name
Alg.Alias.AlgorithmParameters.1.2.840.10040.4.1
CertPathBuilder.PKIX ValidationAlgorithm
Policy.JavaPolicy
Alg.Alias.AlgorithmParameters.1.3.14.3.2.12
Alg.Alias.Signature.SHA/DSA
Alg.Alias.KeyPairGenerator.1.3.14.3.2.12
MessageDigest.SHA-384
Signature.SHA1withDSA ImplementedIn
AlgorithmParameterGenerator.DSA
Signature.NONEwithDSA SupportedKeyClasses
MessageDigest.SHA-512
......
```

在此，本书没有将控制台获得的信息全部展示出来。在Java 5以前，上述提供者的信息还不到1张A4纸，但到了Java 6后，提供者的信息变得相当庞大。如果将上述完整的信息打印到A4纸上，可以密密麻麻地打印出20多页。这同样说明，在Java 6的环境中，对于加密算法的选择有了更广泛的空间。

3.2.3 MessageDigest

MessageDigest类实现了消息摘要算法，它继承于MessageDigestSpi类，是Java安全提供者体系结构中最为简单的一个引擎类。在本章后续的内容中，我们还将看到更多引擎类。

```
// 实现创建和验证消息摘要的操作。
public abstract class MessageDigest
extends MessageDigestSpi
```

在Java API的列表中，我们总能看到后缀名带有SPI（Service Provider Interface，服务提供者接口）的类，它们总是与引擎类形影不离。例如MessageDigestSpi类是MessageDigest 类的服务提供者接口。如果要实现自定义的消息摘要算法，则需要提供继承于MessageDigestSpi类的子类实现。MessageDigest类自然是Sun提供给我们的一个很好的用于实现自定义算法的范例。在其他引擎类的实现中（除签名算法类Signature类以外），不再继承相对应的SPI类。本书以Java 6所提供的算法为准，不计划介绍如何实现自定义算法。如果读者对SPI类有兴趣，可参照Java API中的相应内容。

❑ 方法详述

我们可以通过以下方法获得MessageDigest类的实例：

更多资源请访问稀酷客(www.ckook.com)

```
// 返回实现指定摘要算法的 MessageDigest对象。
public static MessageDigest getInstance(String algorithm)
// 返回实现指定摘要算法的 MessageDigest对象。
public static MessageDigest getInstance(String algorithm, Provider provider)
// 返回实现指定摘要算法的 MessageDigest对象。
public static MessageDigest getInstance(String algorithm, String provider)
```

目前Java 6支持MD2、MD5、SHA-1（SHA）、SHA -256、SHA-384、SHA-512六种消息摘要算法，算法名不区分大小写。

在本书后续的内容中，诸如上述的算法提供者参数（参数Provider provider或参数String provider）会有很多，其值可以为new com.sun.crypto.provider.SunJCE()或"SunJCE"。当使用第三方安全提供者时，需指明provider参数。如果使用BouncyCastleProvider，其值必须为new org.bouncycastle.jce.provider.BouncyCastleProvider()或"BC"。当然，使用第三方安全提供者的前提是需要配置%JDK_HOME%\jre\lib\security\java.security文件。相关内容请参见3.2.1节。

在获得MessageDigest实例化对象后，我们可以利用以下方法对此对象进行操作：

```
// 使用指定的字节更新摘要。
public void update(byte input)
// 使用指定的字节数组更新摘要。
public void update(byte[] input)
```

上述方法参数不同，一个是更新单字节，一个是更新字节数组。

我们也可以使用如下方法，根据偏移量更新摘要：

```
// 使用指定的字节数组，从指定的偏移量开始更新摘要。
public void update(byte[] input, int offset, int len)
```

当然，我们也可以使用缓冲模式更新摘要，可使用如下方法：

```
// 使用指定的字节缓冲更新摘要。
public void update(ByteBuffer input)
```

更新摘要信息，其参数可以是更新一个字节、一个字节数组，甚至是字节数组中的某一段偏移量，也可以是字节缓冲对象。

这些方法的调用顺序不受限制，在向摘要中增加所需数据时可以多次调用。

在完成摘要更新后，我们可以通过以下方法完成摘要操作：

```
// 通过执行诸如填充之类的最终操作完成摘要计算，返回消息摘要字节数组。
public byte[] digest()
```

作者通常会选择上述方法完成摘要处理，也就是在调用上述方法前使用update()方法完成摘要更新后直接使用digest()方法。如果我们只需要执行一次update()方法，不如考虑在digest()方法中直接传入待摘要信息，也就是直接使用如下方法：

```
/* 使用指定的字节数组对摘要进行最后更新，然后完成摘要计算，返回消息摘要字节数组。*/
public byte[] digest(byte[] input)
/* 通过执行诸如填充之类的最终操作完成摘要计算，将消息摘要结果按照指定的偏移量存放在字节数组中，返回消息摘要的长度。*/
```

更多资源请访问稀酷客(www.ckook.com)

```
public int digest(byte[] buf, int offset, int len)
```

完成摘要操作后，可以通过以下方法对比两个摘要信息：

```
// 比较两个摘要的相等性。
public static boolean isEqual(byte[] digesta, byte[] digestb)
```

当两个摘要中的每个字节都相同，且两个摘要都有相同的长度时，则认为两个摘要相同。

```
// 重置摘要以供再次使用。
public void reset()
```

执行该重置方法等同于创建一个新的MessageDigest实例化对象。

我们完成了摘要处理，获得了摘要信息，也就能够获得摘要信息的长度，如果需要这样的长度信息，可以使用如下方法：

```
/* 返回以字节为单位的摘要长度，如果提供者不支持此操作并且实现是不可复制的，则返回0。*/
public final int getDigestLength()
```

除了上述方法外，还常用到以下几个方法：

```
// 返回算法名称，如MD5。
public final String getAlgorithm()
// 返回此信息摘要对象的提供者。
public final Provider getProvider()
```

MessageDigest类覆盖了Object的如下方法：

```
// 返回此信息摘要对象的字符串表示形式。
public String toString()
//如果实现是可复制的，则返回一个副本。
Object clone()
```

❑ 实现示例

我们来做一个简单的摘要处理，如代码清单3-2所示。

代码清单3-2 SHA算法摘要处理

```
// 待做消息摘要算法的原始信息。
byte[] input = "sha".getBytes();
// 初始化MessageDigest对象，将使用SHA算法。
MessageDigest sha = MessageDigest.getInstance("SHA");
// 更新摘要信息。
sha.update(input);
// 获取消息摘要结果。
byte[] output = sha.digest();
```

关于消息摘要算法的实现

MessageDigest、DigestInputStream、DigestOutputStream、Mac均是消息认证技术的引擎类。MessageDigest提供核心的消息摘要实现；DigestInputStream、DigestOutputStream提供以

更多资源请访问稀酷客(www.ckook.com)

MessageDigest为核心的消息摘要流实现；Mac提供基于秘密密钥的安全消息摘要实现。Mac与MessageDigest无任何依赖关系。

除了MessageDigest类提供的MD和SHA两大类算法外，我们知道还有一种摘要算法——Mac，它的算法实现是通过Mac类（javax.crypto.Mac）来实现的，具体Java API内容请参见下一节内容。

3.2.4 DigestInputStream

通过MessageDigest类，我们实现了消息摘要算法，但是通过字节或字节数组的方式完成摘要操作，使用起来并不是很方便，因此有了消息摘要流，包含消息摘要输入流和消息摘要输出流。

DigestInputStream类继承了FilterInputStream类，可以通过读取输入流的方式完成摘要更新，因此我们称它为消息摘要输入流，在指定的读操作方法内部完成MessageDigest类的update()方法。

```
// 提供一个输入流，针对所有通过该流的数据计算出相应的消息摘要。
public class DigestInputStream
extends FilterInputStream
```

❑ 方法详述

可以通过以下方法实例化对象：

```
/* 使用指定的InputStream和MessageDigest创建一个DigestInputStream实例。*/
public DigestInputStream(InputStream stream, MessageDigest digest)
```

当需要更新MessageDigest对象时，可以使用如下方法：

```
// 将指定的MessageDigest与此流相关联。
public void setMessageDigest(MessageDigest digest)
```

与之相对应的，获得MessageDigest对象可以用以下方法：

```
// 返回与此流关联的MessageDigest。
public MessageDigest getMessageDigest()
```

当通过下述方法关闭摘要功能后，DigestInputStream就变成了一般的输入流：

```
// 开启或关闭摘要功能。
public void on(boolean on)
```

请注意要使用下述方法更新摘要信息时，一定要确保DigestInputStream开启摘要功能：

```
// 读取字节并更新消息摘要（如果开启了摘要功能）。
public int read()
// 读入byte数组并更新消息摘要（如果开启了摘要功能）。
public int read(byte[] b, int off, int len)
```

上述方法将调用MessageDigest的update()方法完成摘要更新，在此之后可以通过getMessageDigest()方法获得MessageDigest对象，并执行MessageDigest对象的digest()方法完成

更多资源请访问稀酷客(www.ckook.com)

摘要操作。

摘要DigestInputStream类的相关源码，如代码清单3-3所示。

代码清单3-3 DigestInputStream类读操作部分源代码

```
public int read() throws IOException {
    int ch = in.read();
    if (on && ch != -1) {
        digest.update((byte)ch);
    }
    return ch;
}
public int read(byte[] b, int off, int len) throws IOException {
    int result = in.read(b, off, len);
    if (on && result != -1) {
        digest.update(b, off, result);
    }
    return result;
}
```

除上述方法外，通常还用到以下方法：

```
// 打印此摘要输入流及其关联的消息摘要对象的字符串表示形式。
public String toString()
```

❑ 实现示例

我们可以通过如下方式使用该消息摘要输入流，如代码清单3-4所示。

代码清单3-4 MD5算法摘要输入流处理

```
// 待做消息摘要操作的原始信息。
byte[] input = "md5".getBytes();
// 初始化MessageDigest对象，将使用MD5算法。
MessageDigest md = MessageDigest.getInstance("MD5");
// 构建DigestInputStream对象。
DigestInputStream dis = new DigestInputStream(new ByteArrayInputStream(input), md);
// 流输入
dis.read(input, 0, input.length);
// 获得摘要信息
byte[] output = dis.getMessageDigest().digest();
// 关闭流
dis.close();
```

3.2.5 DigestOutputStream

与DigestInputStream类相对应，DigestOutputStream类继承了FilterOutputStream类，可以通过写入输出流的方式完成摘要更新，因此我们称它为消息摘要输出流，在指定的写操作方法内部完成MessageDigest类的update()方法。

更多资源请访问稀酷客(www.ckook.com)

```
// 提供一个输出流，针对所有通过该流的数据计算出相应的消息摘要。
public class DigestOutputStream
extends FilterOutputStream
```

❑ 方法详述

可以通过以下方法实例化对象：

```
/* 使用指定的OutputStream和MessageDigest创建一个DigestOutputStream实例。*/
public DigestOutputStream(OutputStream stream, MessageDigest digest)
```

以下方法与DigestInputStream类相同：

```
// 将指定的MessageDigest与此流相关联。
public void setMessageDigest(MessageDigest digest)
// 返回与此流关联的MessageDigest。
public MessageDigest getMessageDigest()
```

当通过下述方法关闭摘要功能后，DigestOutputStream就变成了一般的输出流：

```
// 开启或关闭摘要功能。
public void on(boolean on)
```

请注意要使用下述方法更新摘要信息时，一定要确保DigestOutputStream开启摘要功能：

```
/* 使用指定的字节更新消息摘要（如果开启了摘要功能），并将字节写入输出流（不管是否开启了摘要功能）。*/
public void write(int b)
/* 使用指定的子数组更新消息摘要（如果开启了摘要功能），并将子数组写入输出流（不管是否开启了摘要功能）。*/
public void write(byte[] b, int off, int len)
```

上述方法将调用MessageDigest类的update()方法完成摘要更新，在此之后可以通过getMessageDigest()方法获得MessageDigest对象，并执行MessageDigest类的digest()方法完成摘要操作。

摘要DigestOutputStream类的相关源码如代码清单3-5所示。

代码清单3-5　DigestOutputStream类写操作部分源代码

```
public void write(int b) throws IOException {
    if (on) {
        digest.update((byte)b);
    }
    out.write(b);
}
public void write(byte[] b, int off, int len) throws IOException {
    if (on) {
        digest.update(b, off, len);
    }
    out.write(b, off, len);
}
```

除上述方法外，通常还用到以下方法：

```
// 打印此摘要输出流及其关联的消息摘要对象的字符串表示形式。
```

更多资源请访问稀酷客(www.ckook.com)

```
public String toString()
```

❑ 实现示例

我们可以通过如代码清单3-6所示的方式使用该消息摘要输出流。

代码清单3-6 MD5算法摘要输出流处理

```
// 待做消息摘要的原始信息。
byte[] input = "md5".getBytes();
// 初始化MessageDigest，将使用MD5算法。
MessageDigest md = MessageDigest.getInstance("MD5");
// 初始化DigestOutputStream。
DigestOutputStream dos = new DigestOutputStream(new ByteArrayOutputStream(), md);
// 流输出
dos.write(input, 0, input.length);
// 获得摘要信息
byte[] output = dos.getMessageDigest().digest();
// 清空流
dos.flush();
// 关闭流
dos.close();
```

3.2.6 Key

Key接口是所有密钥接口的顶层接口，一切与加密解密有关的操作都离不开Key接口。

```
/* 模型化密钥的概念。由于密钥必须在不同实体之间传输，因此所有的密钥都必须是可以序列化的。*/
public interface Key
extends Serializable
```

❑ 方法详述

所有的密钥都具有三个特征：

• 算法

这里指的是密钥的算法，如DES和DSA等，通过getAlgorithm()方法可获得算法名。

• 编码形式

这里指的是密钥的外部编码形式，密钥根据标准格式（如X.509 SubjectPublicKeyInfo或PKCS#8）编码，使用getEncoded()方法可获得编码格式。

• 格式

这里指的是已编码密钥的格式的名称，使用getFormat()方法可获得格式。

对应上述三个特征，Key接口提供如下三个方法：

```
// 返回此密钥的标准算法名称。
public String getAlgorithm()
/* 返回基本编码格式的密钥，通常为二进制格式，如果此密钥不支持编码，则返回 null。*/
public byte[] getEncoded()
// 返回此密钥的基本编码格式，如果此密钥不支持编码，则返回 null。
public String getFormat()
```

更多资源请访问稀酷客(www.ckook.com)

SecretKey、PublicKey、PrivateKey三大接口继承于Key接口，定义了对称密钥和非对称密钥接口。

1. SecretKey

SecretKey（javax.crypto.SecretKey）接口是对称密钥顶层接口。DES、AES等多种对称密码算法密钥均可通过该接口提供，PBE（javax.crypto.interfaces.PBE）接口提供PEB算法定义并继承了该接口。Mac算法实现过程中，通过SecretKey接口提供秘密密钥。通常使用的是SecretKey接口的实现类SecretKeySpec（javax.crypto.spec.SecretKeySpec）。

```
// 秘密（对称）密钥
public interface SecretKey
extends Key
```

2. PublicKey和PrivateKey

PublicKey、PrivateKey接口是非对称密钥顶层接口。

```
// 公钥
public interface PublicKey
extends Key
// 私钥
public interface PrivateKey
extends Key
```

DH、RSA、DSA和EC等多种非对称密钥接口均继承了这两个接口。RSA、DSA、EC接口均在java.security.interfaces包中。DH的密钥接口位于javax.crypto.interfaces包中。有关于java.security.interfaces包和javax.crypto.interfaces包本章不做细节介绍，有兴趣的读者可以关注相应的Java API内容。

3.2.7 AlgorithmParameters

AlgorithmParameters类是一个引擎类，它提供密码参数的不透明表示。

不透明表示与透明表示的区别在于：

- 不透明表示：在这种表示中，不可以直接访问各参数域，只能得到与参数集相关联的算法名及该参数集的某类编码。
- 透明表示：用户可以通过相应规范类中定义的某个“get”方法来分别访问每个值。

```
// 提供密码参数不透明表示。
public class AlgorithmParameters
extends Object
```

❑ 方法详述

可以通过getInstance()工厂方法实例化AlgorithmParameters对象，如下所示：

```
通常，我们使用如下方法获得实例化对象：
// 返回指定算法的AlgorithmParameters对象。
public static AlgorithmParameters getInstance(String algorithm)
```

更多资源请访问稀酷客(www.ckook.com)

当然，我们可以同时指定算法的相应提供者来完成实例化操作，可使用如下方法：

```
// 返回指定算法的AlgorithmParameters对象。
public static AlgorithmParameters getInstance(String algorithm, Provider provider)
// 返回指定算法的AlgorithmParameters对象。
public static AlgorithmParameters getInstance(String algorithm, String provider)
```

完成对象实例化后，需要对其进行初始化：

```
// 使用在 paramSpec 中指定的参数初始化此AlgorithmParameters对象。
public final void init(AlgorithmParameterSpec paramSpec)
//根据参数的基本解码格式导入指定的参数字节数组并对其解码。
public final void init(byte[] params)
//根据指定的解码方案从参数字节数组导入参数并对其解码。
public final void init(byte[] params, String format)
```

AlgorithmParameters对象只能被初始化一次，无法重用。

在上述init()方法中，我们看到AlgorithmParameterSpec参数，它是加密参数的（透明）规范接口，位于java.security.spec包中，其接口内部无任何方法，仅用于将所有参数规范分组，并为其提供类型安全。所有参数规范都必须实现此接口。

在完成初始化后，可通过以下方法获得参数对象的规范：

```
// 返回此参数对象的（透明）规范。
public final <T extends AlgorithmParameterSpec> T getParameterSpec(Class<T> paramSpec)
```

我们可以通过以下方法获得算法参数：

```
// 返回基本编码格式的参数。
public final byte[] getEncoded()
// 返回以指定方案编码的参数。
public final byte[] getEncoded(String format)
```

此外，AlgorithmParameters类提供了以下常用方法：

```
// 返回与此参数对象关联的算法的名称。
public final String getAlgorithm()
// 返回此参数对象的提供者。
public final Provider getProvider()
// 返回描述参数的格式化字符串。
public final String toString()
```

❑ 实现示例

我们通过代码清单3-7来展示如何使用AlgorithmParameters类获得算法参数。

代码清单3-7 构建DES算法参数

```
// 实例化AlgorithmParameters，并指定DES算法。
AlgorithmParameters ap = AlgorithmParameters.getInstance("DES");
// 使用BigInteger生成参数字节数组。
ap.init(new BigInteger("190506197664891634724 69").toByteArray());
// 获得参数字节数组。
```

更多资源请访问稀酷客(www.ckook.com)

```
byte[] b = ap.getEncoded();
/* 打印经过BigInteger处理后的字符串，将会得到与"190506197664891634724 69"相同的字符串*/
System.out.println(new BigInteger(b).toString());
```

上述字符串（190506197664891634724 69）是作者通过AlgorithmParameterGenerator类操作得到的，并不是随意填写的一个数值。

3.2.8 AlgorithmParameterGenerator

AlgorithmParameterGenerator类用于生成将在某个特定算法中使用的参数集合，我们把它称为算法参数生成器，它同样是一个引擎类。

```
// 生成制定算法的参数集合。
public class AlgorithmParameterGenerator
extends Object
```

❑ 方法详述

AlgorithmParameterGenerator类同样需要通过getInstance()工厂方法完成实例化对象。通过算法名称直接指定算方法参数生成器是最简便的实例化方法了，方法如下所示：

```
/* 返回生成与指定算法一起使用的参数集的AlgorithmParameterGenerator对象。*/
public static AlgorithmParameterGenerator getInstance(String algorithm)
```

或者，指定算法的同时指明该算法的提供者，方法如下所示：

```
/* 返回生成与指定算法一起使用的参数集的AlgorithmParameterGenerator对象。*/
public static AlgorithmParameterGenerator getInstance(String algorithm, Provider provider)
/* 返回生成与指定算法一起使用的参数集的AlgorithmParameterGenerator对象。*/
public static AlgorithmParameterGenerator getInstance(String algorithm, String provider)
```

算法生成器共有两种初始化方式：与算法无关的方式或特定于算法的方式。

两种方式的唯一区别在于对象的初始化：

• 与算法无关的初始化

所有参数生成器都共享“大小”概念和一个随机源。AlgorithmParameterGenerator类提供了两个init ()方法，均包含参数int size，另一个方法则要求提供SecureRandom参数。使用这一方法时，特定于算法的参数生成值（如果有）默认为某些标准值，除非它们可以从指定大小派生。

• 特定于算法的初始化

使用特定于算法的语义初始化参数生成器对象，这些语义由特定于算法的参数生成值集合表示。有两个initialize()方法具有AlgorithmParameterSpec参数，其中一个方法还有一个SecureRandom参数，而另一个方法使用以最高优先级安装的提供者的SecureRandom实现作为随机源。（如果任何安装的提供者都不提供SecureRandom的实现，则使用系统提供的随机源。）

与算法无关的初始化方法如下：

```
// 针对某个特定大小初始化此AlgorithmParameterGenerator对象。
public final void init(int size)
// 针对某个特定大小和随机源初始化此AlgorithmParameterGenerator对象。
public final void init(int size, SecureRandom random)
```

更多资源请访问稀酷客(www.ckook.com)

特定于算法的初始化方法如下：

```
/* 利用特定于算法的参数生成值集合初始化此AlgorithmParameterGenerator对象。*/
public final void init(AlgorithmParameterSpec genParamSpec)
/* 利用特定于算法的参数生成值集合初始化此AlgorithmParameterGenerator对象。*/
public final void init(AlgorithmParameterSpec genParamSpec, SecureRandom random)
```

在完成实例化对象操作后，我们就可以通过以下方法生成算法参数对象了：

```
// 生成AlgorithmParameters对象。
public final AlgorithmParameters generateParameters()
```

此外，算法生成器还提供以下常用方法：

```
// 返回与此参数生成器关联的算法的标准名称。
public final String getAlgorithm()
// 返回此算法参数生成器对象的提供者。
public final Provider getProvider()
```

❑ 实现示例

在3.2.7节中，本书曾提到用于AlgorithmParameter初始化的字符串（190506197664891634724 69）是通过AlgorithmParameterGenerator操作来生成的。代码清单3-8就是获得该字符串的代码：

代码清单3-8 通过算法参数生成器获得DES算法相应的算法参数

```
// 实例化AlgorithmParameterGenerator对象，并指定DES算法。
AlgorithmParameterGenerator apg = AlgorithmParameterGenerator.getInstance("DES");
// 初始化。
apg.init(56);
// 生成AlgorithmParameters对象。
AlgorithmParameters ap = apg.generateParameters();
// 获得参数字节数组。
byte[] b = ap.getEncoded();
/* 打印经过BigInteger处理后的字符串，该字符串就是"19050619766489163472469"*/
System.out.println(new BigInteger(b).toString());
```

当使用Java提供的加密组件时，很少会用到AlgorithmParameters和AlgorithmParameterGenerator两个类，当对算法参数要求极为严格的情况下才会考虑使用这种方式。

3.2.9 KeyPair

KeyPair类是对非对称密钥的扩展，它是密钥对的载体，我们把它称为密钥对。

```
// 建立一个包括公钥和私钥的数据对象。
public final class KeyPair
extends Object
implements Serializable
```

KeyPair类包含两个信息：公钥和私钥。

更多资源请访问稀酷客(www.ckook.com)

我们可以像使用一般数据对象一样使用它。

❑ 方法详述

构造一个KeyPair对象很简单，以下是它的构造方法：

```
// 根据给定的公钥和私钥构造KeyPair对象。
public KeyPair(PublicKey publicKey, PrivateKey privateKey)
```

KeyPair只能通过构造方法初始化内部的公钥和私钥，此外不提供设置公钥和私钥的方法，仅提供获取公钥和私钥的方法：

```
// 返回对此KeyPair对象的公钥组件的引用。
public PublicKey getPublic()
// 返回对此KeyPair对象的私钥组件的引用。
public PrivateKey getPrivate()
```

KeyPair通常由KeyPairGenerator的generateKeyPair()方法来提供。

关于KeyPair对象的生成实现，请关注3.2.10节的内容。

3.2.10 KeyPairGenerator

公钥和私钥的生成是由KeyPairGenerator类来实现的，因此我们把它称为密钥对生成器，它同样是一个引擎类。

```
// 生成用于非对称加密算法中含有公钥/私钥的密钥对，并提供相关信息。
public abstract class KeyPairGenerator
extends KeyPairGeneratorSpi
```

在Java 6中，提供了DH、DSA和RSA等多种非对称加密算法实现。

关于Java 6中提供的算法信息，请查看本书附录。

❑ 方法详述

KeyPairGenerator本身是一个抽象类，可通过getInstance()工厂方法实例化对象。

通常我们指定算法名称，直接获得密钥对实例化对象，方法如下所示：

```
// 返回生成指定算法的公钥/私钥密钥对的KeyPairGenerator对象。
public static KeyPairGenerator getInstance(String algorithm)
```

或者，指定算法的同时指明该算法的提供者，方法如下所示：

```
// 返回生成指定算法的公钥/私钥密钥对的KeyPairGenerator对象。
public static KeyPairGenerator getInstance(String algorithm, Provider provider)
// 返回生成指定算法的公钥/私钥密钥对的KeyPairGenerator对象。
public static KeyPairGenerator getInstance(String algorithm, String provider)
```

密钥对生成器共有两种初始化方式：与算法无关的方式和特定于算法的方式，这一点类似于算法参数生成器。

两种方式的唯一区别在于对象的初始化：

• 与算法无关的初始化

所有的密钥对生成器遵循密钥大小和随机源的概念。KeyPairGenerator类提供了两个initialize()

更多资源请访问稀酷客(www.ckook.com)

方法，这两个方法都含有密钥大小参数，另一个方法还要求提供SecureRandom参数。

• 特定于算法的初始化

对于特定于算法的参数集合已存在的情况（例如，DSA中所谓的公用参数），KeyPairGenerator类提供了两个initialize()方法，都具有AlgorithmParameterSpec参数。其中一个方法还有一个SecureRandom参数，而另一个方法使用以最高优先级安装的提供者的 SecureRandom实现作为随机源。（如果任何安装的提供者都不提供SecureRandom的实现，则使用系统提供的随机源。）

与算法无关的初始化方法如下：

```
/*  初始化给定密钥大小的密钥对生成器，使用默认的参数集合，并使用以最高优先级安装的提供者的
SecureRandom 实现作为随机源。*/
public void initialize(int keysize)
// 使用给定的随机源（和默认的参数集合）初始化确定密钥大小的密钥对生成器。
public void initialize(int keysize, SecureRandom random)
```

特定于算法的初始化方法如下：

```
/*  初始化KeyPairGenerator对象，使用指定参数集合，并使用以最高优先级安装的提供者的
SecureRandom 的实现作为随机源。*/
public void initialize(AlgorithmParameterSpec params)
// 使用给定参数集合和随机源初始化KeyPairGenerator对象。
public void initialize(AlgorithmParameterSpec params, SecureRandom random)
```

通常，较为常用的是前两个方法。注意参数int keysize，这个参数用来控制密钥长度，单位为位。

我们通过第2章的学习已经知道，密钥长度决定密码安全强度。在使用该类实现具体某一种算法时，需要切合实际设置相应的密钥长度。如构建DH算法密钥对，其密钥长度要求为64位的倍数，密钥长度范围为512～1024位，即DH算法密钥长度可为512、576、640位等，按64位的整数倍递增至1024位。

如果在实例化操作后未执行初始化操作，密钥长度将设置为默认长度。如DH算法密钥长度默认值为1024位。关于Java 6中密码算法与密钥长度说明，请见本书附录。

在完成上述操作后，可以通过如下方法生成密钥对：

```
// 生成一个KeyPair对象。
public KeyPair generateKeyPair()
// 生成KeyPair对象。
Public final KeyPair genKeyPair()
```

注意这两个方法，genKeyPair()方法内部实现时调用了generateKeyPair()方法，给出相应的代码片段：

```
public KeyPair generateKeyPair() {
    return null;
}
public final KeyPair genKeyPair() {
    return generateKeyPair();
}
```

更多资源请访问稀酷客(www.ckook.com)

generateKeyPair()方法由具体的密钥对生成器提供者实现，通常使用genKeyPair()方法获得密钥对。

与其他引擎类一样，密钥对生成器提供如下两种方法：

```
// 返回此密钥对生成器算法的标准名称。
public String getAlgorithm()
// 返回此密钥对生成器对象的提供者。
public final Provider getProvider()
```

❑ 实现示例

经过上述方法相结合，我们可以很方便地生成一个密钥对，如生成一个DSA算法密钥对：

```
// 实例化KeyPairGenerator对象。
KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA");
// 初始化KeyPairGenerator对象。
kpg.initialize(1024);
// 生成KeyPair对象。
KeyPair keys = kpg.genKeyPair();
```

KeyPairGenerator类提供了非对称密钥对的生成实现，如果要生成私钥可使用KeyGenerator（javax.crypto.KeyGenerator）类。我们将在下一节中讲述它。

3.2.11 KeyFactory

KeyFactory类也是用来生成密钥（公钥和私钥）的引擎类，我们将它称为密钥工厂，它用来生成公钥/私钥，或者说它的作用是通过密钥规范还原密钥。与KeyFactory类相对应的类是SecretKeyFactory类，这个类用于生成秘密密钥，我们将在3.3.4节中详述。KeySpec接口定义了密钥规范，有关KeySpec接口，请参见3.4.1节。

```
// 按照指定的编码格式或密钥参数，提供一个用于输入和输出密钥的基础设施。
public class KeyFactory
extends Object
```

❑ 方法详述

KeyFactory类同样通过getInstance()工厂方法来获得实例化对象。

我们可以通过指定算法名称的方式获得实例化对象，如下所示：

```
// 返回转换指定算法的公钥/私钥关键字的KeyFactory对象。
public static KeyFactory getInstance(String algorithm)
```

另外一种方式就是指定算法名称的时候指定该算法的提供者，如下所示：

```
// 返回转换指定算法的公钥/私钥关键字的KeyFactory对象。
public static KeyFactory getInstance(String algorithm, Provider provider)
// 返回转换指定算法的公钥/私钥关键字的KeyFactory对象。
public static KeyFactory getInstance(String algorithm, String provider)
```

KeyFactory类最常用的操作就是通过密钥规范获得相应的密钥，其方法如下：

更多资源请访问稀酷客(www.ckook.com)

```
// 根据提供的密钥规范（密钥材料）生成PublicKey对象。
public final PublicKey generatePublic(KeySpec keySpec)
// 根据提供的密钥规范（密钥材料）生成PrivateKey对象。
public final PrivateKey generatePrivate(KeySpec keySpec)
```

可以通过以下方法获得密钥规范：

```
// 返回给定密钥对象的规范（密钥材料）。
public final <T extends KeySpec> T getKeySpec(Key key, Class<T> keySpec)
```

可以通过以下方法转换来自不同提供者装载的密钥类型：

```
// 将提供者可能未知或不受信任的密钥对象转换成此密钥工厂对应的密钥对象。
public final Key translateKey(Key key)
```

密钥工厂同样提供以下两个常用方法：

```
// 获取与此 KeyFactory关联的算法的名称。
public final String getAlgorithm()
// 返回此KeyFactory对象的提供者。
public final Provider getProvider()
```

❑ 实现示例

通过代码清单3-9来说明密钥工厂如何将密钥字节数组转换为密钥对象。

代码清单3-9 构建密钥对与还原密钥

```
// 实例化KeyPairGenerator对象，并指定RSA算法。
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("RSA");
// 初始化KeyPairGenerator对象。
keyPairGen.initialize(1024);
// 生成KeyPair对象。
KeyPair keyPair = keyPairGen.generateKeyPair();
// 获得私钥密钥字节数组。实际使用过程中该密钥以此种形式保存传递给另一方。
byte[] keyBytes = keyPair.getPrivate().getEncoded();
// 由私钥密钥字节数组获得密钥规范。
PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(keyBytes);
// 实例化密钥工厂，并指定RSA算法。
KeyFactory keyFactory = KeyFactory.getInstance("RSA");
// 生成私钥
Key privateKey = keyFactory.generatePrivate(pkcs8KeySpec);
```

有关密钥规范请参见3.4节的相关内容。

3.2.12 SecureRandom

SecureRandom类继承于Random类（java.util.Random），它起到强加密随机数生成器（Random Number Generator，RNG）的作用，我们称它为安全随机数生成器，它同样是一个引擎类。

```
// 提供安全随机数。
```

更多资源请访问稀酷客(www.ckook.com)

```
publicclass SecureRandom
extends Random
```

❑ 方法详述

安全随机数生成器可通过以下构造方法进行实例化对象：

```
// 构造一个实现默认随机数算法的SecureRandom对象。
public SecureRandom()
// 在给定种子的前提下，构造一个实现默认随机数算法的SecureRandom对象。
public SecureRandom(byte[] seed)
```

当然，我们可以通过给定的其他参数，使用getInstance()工厂方法来完成实例化对象。

我们可以通过指定算法名称的方式获得安全随机数对象，方法如下所示：

```
// 返回实现指定随机数生成器算法的SecureRandom对象。
public static SecureRandom getInstance(String algorithm)
```

或者，指明该算法名的同时指定该算法的提供者：

```
// 返回实现指定随机数生成器算法的SecureRandom对象。
public static SecureRandom getInstance(String algorithm, Provider provider)
// 返回实现指定随机数生成器算法的SecureRandom对象。
public static SecureRandom getInstance(String algorithm, String provider)
```

SHA1PRNG算法是SecureRandom的默认算法。

在获得实例化对象后，可以多次使用如下方法生成种子：

```
/* 返回给定的种子字节数量，该数量可使用此类用来将自身设置为种子的种子生成算法来计算。*/
public byte[] generateSeed(int numBytes)
```

SecureRandom类覆盖了Random类的以下几个方法：

```
// 生成用户指定的随机字节数。其结果将填充在参数byte[] bytes中。
public synchronized void nextBytes(byte[] bytes)
// 使用给定 long seed 中包含的8个字节，重新设置此随机对象的种子。
public void setSeed(long seed)
```

使用如下方法将重置随机数对象中的种子：

```
//重新设置此随机对象的种子。
public synchronized void setSeed(byte[] seed)
```

使用如下方法将获得新的随机数：

```
/* 返回给定的种子字节数量，该数量可使用此类用来将自身设置为种子的种子生成算法来计算。*/
public static byte[] getSeed(int numBytes)
```

此外，安全随机数生成器作为引擎类，提供如下常用方法：

```
// 返回此安全随机数生成器对象实现的算法的名称。
public String getAlgorithm()
// 返回此安全随机数生成器对象的提供者。
public final Provider getProvider()
```

更多资源请访问稀酷客(www.ckook.com)

❑ 实现示例

安全随机数生成器常用来配合生成秘密密钥，如代码清单3-10所示。

代码清单3-10 构建安全随机数对象及秘密密钥对象

```
// 实例化SecureRandom对象
SecureRandom secureRandom = new SecureRandom();
// 实例化KeyGenerator对象
KeyGenerator kg = KeyGenerator.getInstance("DES");
// 初始化KeyGenerator对象
kg.init(secureRandom);
// 生成SecretKey对象
SecretKey secretKey = kg.generateKey();
```

SecureRandom类的另一个用途在数字签名Signature类中辅助完成初始化操作。

3.2.13 Signature

Signature 类用来生成和验证数字签名，它同样是一个引擎类。

```
// 提供一个引擎，用以创建和验证数字签名。
public abstract class Signature
extends SignatureSpi
```

❑ 方法详述

使用Signature对象签名数据或验证签名包括以下三个阶段：

1）初始化

• 初始化验证签名的公钥

• 初始化签署签名的私钥

2）更新

• 根据初始化类型，可更新要签名或验证的字节。

3）签署或验证所有更新字节的签名。

按照以上描述，我们先通过getInstance()工厂方法完成实例化对象。

一种简单的方式就是指定算法名称，直接获得Signature对象，方法如下所示：

```
// 返回实现指定签名算法的 Signature对象。
public static Signature getInstance(String algorithm)
```

另一种方式就是指定算法名称的同时指定该算法的提供者，方法如下所示：

```
// 返回实现指定签名算法的 Signature对象。
public static Signature getInstance(String algorithm, Provider provider)
// 返回实现指定签名算法的 Signature对象。
public static Signature getInstance(String algorithm, String provider)
```

在以下几步操作中，我们会看到消息摘要处理的一些类似的方法，大家可以参见3.2.3节和3.3.1节的内容。

更多资源请访问稀酷客(www.ckook.com)

目前，Signature支持NONEwithDSA和SHA1withDSA两种基于DSA算法的签名算法，同时还支持MD2withRSA、MD5withRSA、SHA1withRSA、SHA256withRSA、SHA384withRSA和SHA512withRSA六种基于RSA算法的签名算法。

关于Java 6版本中提供的算法信息，请参见本书附录。

完成实例化对象后，我们要初始化Signature对象：

```
// 初始化这个用于签名的Signature对象。
public final void initSign(PrivateKey privateKey)
// 初始化这个用于签名的Signature对象。
public final void initSign(PrivateKey privateKey, SecureRandom random)
```

上述两种方法是用于签名操作的初始化，如果用于验证操作则需要通过以下方法：

```
// 初始化此用于验证的Signature对象。
public final void initVerify(PublicKey publicKey)
// 使用来自给定证书的公钥初始化用于验证的Signature对象。
public final void initVerify(Certificate certificate)
```

注意上述方法的参数有所不同，前一种方法用于一般数字签名的验证操作，后一种方法则用于数字证书的验证操作。

完成初始化操作后，我们就可以通过以下方法更新Signature对象中的数据了。

Signature类的update()方法定义与MessageDigest类的update()方法相类似，可以通过更新一个字节，也可以更新一个字节数组，方法如下所示：

```
// 更新要由字节签名或验证的数据。
public final void update(byte b)
// 使用指定的字节数组更新要签名或验证的数据。
public final void update(byte[] data)
```

另一种方式就是根据偏离量做更新处理，方法如下所示：

```
// 从指定的偏移量开始，使用指定的 byte 数组更新要签名或验证的数据。
public final void update(byte[] data, int off, int len)
```

或者使用缓冲方式，方法如下所示：

```
// 使用指定的 ByteBuffer 更新要签名或验证的数据。
public final void update(ByteBuffer data)
```

在完成更新操作后，我们就可以做签名操作了：

```
// 返回所有已更新数据的签名字节。
public final byte[] sign()
// 完成签名操作，并得到存储在缓冲区中的签名字节长度。
public final int sign(byte[] outbuf, int offset, int len)
```

终于到了验证操作这一步，可通过以下方法完成：

```
// 验证传入的签名，并返回验证结果。
public final boolean verify(byte[] signature)
// 从指定的偏移量开始，验证指定的字节数组中传入的签名，并返回验证结果。
```

更多资源请访问稀酷客(www.ckook.com)

```
public final boolean verify(byte[] signature, int offset, int length)
```

此外，Signature提供了以下两种方法来设置和获取算法参数：

```
// 使用指定的参数集初始化此签名引擎。
public final void setParameter(AlgorithmParameterSpec params)
// 返回与此签名对象一起使用的参数。
public final AlgorithmParameters getParameters()
```

Signature类作为引擎类，同样提供如下两个常用方法：

```
// 返回此签名对象的算法名称。
public final String getAlgorithm()
// 返回此签名对象的提供者。
public final Provider getProvider()
```

Signature类覆盖了SignatureSpi类，方法如下所示：

```
// 如果此实现可以复制，则返回副本。
public Object clone()
/* 返回此签名对象的字符串表示形式，以提供包括对象状态和所用算法名称在内的信息。*/
public String toString()
```

❑ 实现示例

代码清单3-11展示了如何使用私钥完成数字签名的操作。

代码清单3-11 数字签名处理

```
// 待做数字签名的原始信息。
byte[] data = "Data Signature".getBytes();
// 实例化KeyPairGenerator对象，并指定DSA算法。
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DSA");
// 初始化KeyPairGenerator对象。
keyPairGen.initialize(1024);
// 生成KeyPair对象。
KeyPair keyPair = keyPairGen.generateKeyPair();
// 实例化Signature对象。
Signature signature = Signature.getInstance(keyPairGen.getAlgorithm());
// 初始化用于签名操作的Signature对象。
signature.initSign(keyPair.getPrivate());
// 更新
signature.update(data);
// 获得签名，即字节数组sign。
byte[] sign = signature.sign();
```

私钥完成签名，公钥则用于完成验证，方法如下所示：

```
// 初始化用于验证操作的Signature对象。
signature.initVerify(keyPair.getPublic());
// 更新
signature.update(data);
// 获得验证结果
```

更多资源请访问稀酷客(www.ckook.com)

```
boolean status = signature.verify(sign);
```

在上述示例代码中，如果变量status值为true，则认为验证成功。上述代码稍加改动就可用于数字证书的签名和认证操作。

3.2.14 SignedObject

SignedObject类是一个用来创建实际运行时对象的类，在检测不到这些对象的情况下，其完整性不会遭受损害。更明确地说，SignedObject包含另外一个Serializable对象，即要签名的对象及其签名，我们可以称它为签名对象。

签名对象是对原始对象的“深层复制”（以序列化形式），一旦生成了副本，对原始对象的进一步操作就不再影响该副本。

```
// 实现对象与数字签名的封装。
public final class SignedObject
extends Object
implements Serializable
```

❑ 方法详述

签名对象通过以下构造方法完成实例化对象：

```
// 通过任何可序列化对象构造 SignedObject对象。
public SignedObject(Serializable object, PrivateKey signingKey, Signature signingEngine)
```

在完成上述实例化操作后，可通过以下方法获得封装后的对象和签名：

```
// 获取已封装的对象。
public Object getObject()
// 在已签名对象上按字节数组的形式获取签名。
public byte[] getSignature()
```

接着，可以通过公钥和Signature进行验证操作：

```
/* 使用指派的验证引擎，通过给定的验证密钥验证SignedObject中的签名是否为内部存储对象的有效签名。*/
public boolean verify(PublicKey verificationKey, Signature verificationEngine)
```

此外，SignedObject还提供了以下方法：

```
// 获取签名算法的名称。
String getAlgorithm()
```

❑ 实现示例

我们对3.2.13节中的签名验证代码稍作改动，如代码清单3-12所示。

代码清单3-12 数字签名对象处理

```
// 待做数字签名的原始信息。
byte[] data = "Data Signature".getBytes();
// 实例化KeyPairGenerator对象，并指定DSA算法。
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DSA");
// 初始化KeyPairGenerator对象。
```

更多资源请访问稀酷客(www.ckook.com)

```
keyPairGen.initialize(1024);
// 生成KeyPair对象
KeyPair keyPair = keyPairGen.generateKeyPair();
// 实例化Signature对象
Signature signature = Signature.getInstance(keyPairGen.getAlgorithm());
```

这里与3.2.13节略有不同：

```
// 实例化SignedObject对象
SignedObject s = new SignedObject(data, keyPair.getPrivate(), signature);
```

我们通过另一种方式得到了签名：

```
// 获得签名值
byte[] sign = s.getSignature();
```

验证方式也有所不同：

```
// 验证签名
boolean status = s.verify(keyPair.getPublic(), signature);
```

在上述示例代码中，如果变量status值为true，则认为验证成功。

3.2.15 Timestamp

Timestamp类用于封装有关签署时间戳的信息，且它是不可更改的。它包括时间戳的日期和时间，以及有关生成和签署时间戳的Timestamping Authority（TSA）的信息。为了区别于一般时间戳（java.sql.Timestamp），我们称它为数字时间戳。

```
// 构建数字时间戳
public final class Timestamp
extends Object
implements Serializable
```

❑ 方法详述

构建一个数字时间戳需要提供时间和签名证书路径（CertPath）两个参数，方法如下：

```
// 构造一个Timestamp对象。
public Timestamp(Date timestamp, CertPath signerCertPath)
```

获得数字时间戳后的主要目的在于校验给定对象是否与此数字时间戳一致，方法如下：

```
// 比较指定的对象和Timestamp对象是否相同。
public boolean equals(Object obj)
```

当然，我们可以通过数字时间戳获得相应的签名证书路径和生成数字时间戳的日期和时间，方法如下：

```
// 返回 Timestamping Authority 的证书路径。
public CertPath getSignerCertPath()
// 返回生成数字时间戳时的日期和时间。
public Date getTimestamp()
```

更多资源请访问稀酷客(www.ckook.com)

此外，数字时间戳覆盖了以下两种方法：

```
// 返回此数字时间戳的散列码值。
public int hashCode()
// 返回描述此数字时间戳的字符串。
public String toString()
```

❑ 实现示例

我们可以通过代码清单3-13所示来构造一个数字时间戳。

代码清单3-13 获得数字时间戳

```
// 构建CertificateFactory对象，并指定证书类型为X.509。
CertificateFactory cf = CertificateFactory.getInstance("X509");
// 生成CertPath对象
CertPath cp = cf.generateCertPath(new FileInputStream("D:\\x.cer"));
// 实例化数字时间戳
Timestamp t = new Timestamp(new Date(), cp);
```

关于CertificateFactory类的介绍请参见3.5.1节。

3.2.16 CodeSigner

CodeSigner类封装了代码签名者的信息，且它是不可变的，我们称它为代码签名。它和数字时间戳（java.security.Timestamp）紧密相连。

```
// 用于构建代码签名
public final class CodeSigner
extends Object
implements Serializable
```

❑ 方法详述

CodeSigner类是一个终态类，可以通过其构造方法完成实力化对象：

```
// 构造CodeSigner对象
public CodeSigner(CertPath signerCertPath, Timestamp timestamp)
```

完成实例化对象后，就可以通过以下方法获得其属性：

```
// 返回签名者的CertPath对象
public CertPath getSignerCertPath()
// 返回签名时间戳
public Timestamp getTimestamp()
```

注意，这里的Timestamp是java.security.Timestamp，是用做数字时间戳的Timestamp！

获得CodeSigner对象后的最重要的操作就是执行比对，CodeSigner覆盖了equals()方法：

```
// 测试指定对象与此CodeSigner对象是否相等。
public boolean equals(Object obj)
```

如果，传入参数不是CodeSigner类的实现，则直接返回false。如果，传入参数是CodeSigner类的实现，则比较其Timestamp和CertPath两个属性，此方法的实现如代码清单3-14所示。

更多资源请访问稀酷客(www.ckook.com)

代码清单3-14 CodeSigner类equals()方法源代码片段

```
private CertPath signerCertPath;
private Timestamp timestamp;
// ......
public boolean equals(Object obj) {
    if (obj == null || (!(obj instanceof CodeSigner))) {
            return false;
    }
    CodeSigner that = (CodeSigner)obj;
    if (this == that) {
            return true;
    }
    Timestamp thatTimestamp = that.getTimestamp();
    if (timestamp == null) {
            if (thatTimestamp != null) {
                    return false;
            }
    } else {
            if (thatTimestamp == null ||
                    (! timestamp.equals(thatTimestamp))) {
                    return false;
            }
    }
            return signerCertPath.equals(that.getSignerCertPath());
}
```

此外，CodeSigner还覆盖了以下两种方法：

```
// 返回此代码签名者的散列码值。
public int hashCode()
// 返回描述此代码签名者的字符串。
public String toString()
```

❑ 实现示例

我们接3.2.15节的代码实现，来构建一个代码签名，如代码清单3-15所示。

代码清单3-15 验证代码签名

```
// 构建CertificateFactory对象，并指定证书类型为X.509。
CertificateFactory cf = CertificateFactory.getInstance("X509");
// 生成CertPath对象
CertPath cp = cf.generateCertPath(new FileInputStream("D:\\x.cer"));
// 实例化Timestamp对象
Timestamp t = new Timestamp(new Date(), cp);
// 实例化CodeSigner对象
CodeSigner cs = new CodeSigner(cp, t);
// 获得比对结果
boolean status = cs.equals(new CodeSigner(cp, t));
```

得到CodeSigner对象后，可以使用它的equals()方法来进行比对。

更多资源请访问稀酷客(www.ckook.com)

3.2.17 KeyStore

KeyStore类被称为密钥库，用于管理密钥和证书的存储。KeyStore 类是个引擎类，它提供一些相当完善的接口来访问和修改密钥仓库中的信息。

```
// 用于管理密钥和证书存储。
public class KeyStore
extends Object
```

❑ 方法详述

KeyStore类同样需要通过getInstance()工厂方法获得实例化对象。

我们可以通过指定密钥库类型获得实例化对象，方法如下所示：

```
// 返回指定类型的KeyStore对象。
public static KeyStore getInstance(String type)
```

另一种方式就是指定密钥库类型时指定该密钥库类型的提供者，方法如下所示：

```
// 返回指定类型的KeyStore对象。
public static KeyStore getInstance(String type, Provider provider)
// 返回指定类型的KeyStore对象。
public static KeyStore getInstance(String type, String provider)
```

完成实例化操作后，我们将进行以下操作：

```
/* 返回Java安全属性文件中指定的默认密钥库类型；如果不存在此类属性，则返回字符串"jks" ("Java
keystore" 的首字母缩写)。*/
public final static String getDefaultType()
```

密钥库类型不区分大小。例如，“JKS”被认为与“jks”相同。除了JKS这种类型以外，还有PKCS12和JCEKS两种类型。JCEKS本身受美国出口限制，所以通常我们可以使用的只有JKS和PKCS12两种类型。但PKCS12这种类型的密钥库管理支持不是很完备，只能够读取该类型的证书，却不能对其进行修改。因此，JKS是最好的选择。

与获取默认密钥库类型方法相对的是如下取得当前密钥库类型的方法：

```
// 返回此密钥库的类型。
public final String getType()
```

通过以下两种方法来加载和存储密钥库：

```
// 从给定输入流中加载此密钥库。
public final void load(InputStream stream, char[] password)
// 将此密钥库存储到给定输出流，并用给定密码保护其完整性。
public final void store(OutputStream stream, char[] password)
```

通过以下方法获得密钥库中的条目数：

```
// 获取此密钥库中的条目数。
public final int size()
```

我们依旧可以使用如下方法获得提供者：

更多资源请访问稀酷客(www.ckook.com)

```
// 返回此密钥库的提供者。
public final Provider getProvider()
```

在密钥库中，密钥和证书都是通过别名进行组织的。

可通过以下方法获得密钥库的别名列表，以及确认给定的别名是否在当前密钥库中：

```
// 列出此密钥库的所有别名。
public final Enumeration<String> aliases()
// 检查给定别名是否存在于此密钥库中。
public final boolean containsAlias(String alias)
```

我们可以通过别名和密码来获得密钥：

```
// 返回与给定别名关联的密钥，并用给定密码来恢复它。
public final Key getKey(String alias, char[] password)
```

这里要注意一下，虽然返回的类型是Key接口，但真正获得的是PrivateKey接口的实例。

通过该方法将获得该别名在密钥库中对应的私钥。

通过别名，我们还可以获得证书或证书链：

```
// 返回与给定别名关联的证书。
public final Certificate getCertificate(String alias)
// 返回与给定别名关联的证书链。
public final Certificate[] getCertificateChain(String alias)
```

反之，通过证书获得其在密钥库中的别名：

```
// 返回证书与给定证书匹配的第一个密钥库条目的别名。
public final String getCertificateAlias(Certificate cert)
```

同样，能通过别名获得其对应条码的创建日期：

```
// 返回给定别名标识的条目的创建日期。
public final Date getCreationDate(String alias)
```

通过别名来删除密钥库中与别名相对应的条目：

```
// 删除此密钥库中给定别名标识的条目。
public final void deleteEntry(String alias)
```

我们一直在说“条目”这个词，是因为在密钥库中别名可能对应密钥，有可能对应证书。我们通过以下方法设置别名与密钥和证书的对应关系。

```
// 将给定密钥（受保护的）分配给指定的别名。
public final void setKeyEntry(String alias, byte[] key, Certificate[] chain)
// 将给定的密钥分配给给定的别名，并用给定密码保护它。
public final void setKeyEntry(String alias, Key key, char[] password,
Certificate[] chain)
```

我们也可以使用如下方法将别名与证书绑定：

```
// 将给定可信证书分配给给定别名。
public final void setCertificateEntry(String alias, Certificate cert)
```

既然，别名可能对应密钥或证书，那么需要方法来判别：

更多资源请访问稀酷客(www.ckook.com)

```
/* 如果给定别名标识的条目是通过调用 setCertificateEntry 或者以 TrustedCertificateEntry
为参数的 setEntry 创建的，则返回 true。*/
public final boolean isCertificateEntry(String alias)
/* 如果给定别名标识的条目是通过调用 setKeyEntry 或者以 PrivateKeyEntry 或
SecretKeyEntry 为参数的 setEntry 创建的，则返回 true。*/
public final boolean isKeyEntry(String alias)
```

在上述方法的注释中，我们注意到一些Java 5之前所没有的内容。在Java 5以后，KeyStore类中加入了新的内部接口及内部类。

KeyStore.Entry接口：

```
// 用于密钥库项类型的标记接口。
public static interface KeyStore.Entry
```

KeyStore.Entry接口是一个空接口，内部没有定义代码，用于类型区分。

KeyStore管理不同类型的条目。每种类型的条目都实现KeyStore.Entry接口。提供了三种基本的KeyStore.Entry实现：

```
// 保存私钥和相应证书链的密钥库项。
public static final class KeyStore.PrivateKeyEntry
// 保存秘密密钥的密钥库项。
public static final class KeyStore.SecretKeyEntry
// 保存可信的证书的密钥库项。
public static final class KeyStore.TrustedCertificateEntry
```

- KeyStore.PrivateKeyEntry

我们把KeyStore.PrivateKeyEntry称为私钥项。可通过以下方法获得实例化对象：

```
// 构造带私钥和相应证书链的私钥项。
public KeyStore.PrivateKeyEntry(PrivateKey privateKey, Certificate[] chain)
```

获得私钥项对象后，就可以获得其相应的属性：

```
/* 从此私钥项内部的证书链数组首位中获取证书。如果证书的类型是 X.509，返回证书的运行时类型是
X509Certificate*/
public Certificate getCertificate()
// 从此私钥项获取证书链。
public Certificate[] getCertificateChain()
// 从此私钥项获取PrivateKey对象。
public PrivateKey getPrivateKey()
```

此外，私钥项覆盖了如下方法：

```
// 返回此私钥项的字符串表示形式。
public String toString()
```

- KeyStore.SecretKeyEntry

我们把KeyStore. SecretKeyEntry称为秘密密钥项。可通过以下方法获得实例化对象：

```
// 用秘密密钥构造秘密密钥项。
public KeyStore.SecretKeyEntry(SecretKey secretKey)
```

更多资源请访问稀酷客(www.ckook.com)

秘密密钥项的主要作用就是保护秘密密钥，可通过如下方法获得秘密密钥：

```
// 从此项中获取SecretKey对象。
public SecretKey getSecretKey()
```

此外，秘密密钥项覆盖了如下方法：

```
// 返回此秘密密钥项的字符串表示形式。
public String toString()
```

• KeyStore.TrustedCertificateEntry

我们把KeyStore.TrustedCertificateEntry称为信任证书项，可通过以下方法获得实例化对象：

```
// 用可信证书构造信任证书项。
public KeyStore.TrustedCertificateEntry(Certificate trustedCert)
```

与秘密密钥项相似，信任证书项主要保护受信任的证书，可通过如下方法获得其证书：

```
// 从此信任证书项获取可信证书。
public Certificate getTrustedCertificate()
```

此外，秘密密钥项覆盖了如下方法：

```
// 返回此信任证书项的字符串表示形式。
public String toString()
```

在理解了上述各种KeyStore.Entry的具体实现后，我们来了解一下密钥库中与KeyStore.Entry相关的方法：

```
// 确定指定别名的密钥库项是否是指定密钥库项的实例或子类。
public final boolean entryInstanceOf(String alias,Class<? extends KeyStore.Entry> entryClass)
```

除了上述内部接口与内部实现外，还有其他内部接口及相应的内部实现。由于内容与本书所涉及的方面关联较少，故不在这里详述。有兴趣的读者朋友可以参考相关API文档。

❑ 实现示例

我们通过代码清单3-16来演示如何获得密钥库对象。

代码清单3-16 加载密钥库

```
// 加载密钥库文件
FileInputStream is = new FileInputStream("D:\\.keystore");
// 实例化KeyStore对象
KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());
// 加载密钥库，使用密码"password"。
ks.load(is, "password".toCharArray());
// 关闭文件流
is.close();
```

得到密钥库对象后，可以获得其别名对应的私钥：

```
// 获得别名为"alias"所对应的私钥。
PrivateKey key = (PrivateKey) ks.getKey("alias", "password".toCharArray());
```

更多资源请访问稀酷客(www.ckook.com)

或者使用以下私钥项的方式获得私钥：

```
// 获得私钥项
KeyStore.PrivateKeyEntry pkEntry = (KeyStore.PrivateKeyEntry)ks.getEntry("alias",
"password".toCharArray());
// 获得私钥
PrivateKey privateKey = pkEntry.getPrivateKey();
```

有关证书类Certificate请关注3.5.1节。

3.3 javax.crypto包详解

javax.crypto包为加密操作提供类和接口。在上一节内容中，我们了解了如何实现消息摘要算法、获得对称密钥等API内容。但是，如果缺乏Cipher类，我们将无法完成加密与解密的实现。

在本节内容中，读者将可以完成完整的算法实现，以及安全摘要算法的实现。

3.3.1 Mac

Mac属于消息摘要的一种，但它不同于一般消息摘要（如MessageDigest提供的消息摘要实现），仅通过输入数据无法获得消息摘要，必须有一个由发送方和接收方共享的秘密密钥才能生成最终的消息摘要——安全消息摘要。安全消息摘要也称为消息认证（鉴别）码（Message Authentication Code，Mac）。

在Java 6版本中支持HmacMD5、HmacSHA1、HmacSHA256、HmacSHA384和HmacSHA512算法。

关于Java 6版本中提供的算法信息，请参见本书附录。

```
// 实现创建和验证安全消息摘要的操作。
public class Mac
extends Object
implements Cloneable
```

❑ 方法详述

Mac与MessageDigest绝大多数方法相同，我们可以通过以下方法获得它的实例：

```
// 返回实现指定摘要算法的Mac对象。
public final static Mac getInstance(String algorithm)
```

或者，指定算法名称的同时指定该算法的提供者。

```
// 返回实现指定摘要算法的Mac对象。
public final static Mac getInstance(String algorithm, Provider provider)
// 返回实现指定摘要算法的Mac对象。
public final static Mac getInstance(String algorithm, String provider)
```

目前，Mac类支持HmacMD5、HmacSHA1、HmacSHA256、HmacSHA384、HmacSHA512

更多资源请访问稀酷客(www.ckook.com)

5种消息摘要算法。

在获得Mac实例化对象后，需要通过给定的密钥对Mac对象初始化，方法如下所示：

```
// 用给定的密钥初始化此 Mac。
public final void init(Key key)
// 用给定的密钥和算法参数初始化此 Mac。
public final void init(Key key, AlgorithmParameterSpec params)
```

这里的密钥Key指的是秘密密钥，请使用该密钥作为init()方法的参数。

Mac类更新摘要方法与MessageDigest类相同，方法如下所示：

```
// 使用指定的字节更新摘要。
public final void update(byte input)
// 使用指定的字节数组更新摘要。
public final void update(byte[] input)
```

上述方法传入参数不同，前一个传入的是字节，后一个传入的是字节数组。

我们也可以通过输入偏移量的方式做更新操作，方法如下所示：

```
// 使用指定的字节数组，从指定的偏移量开始更新摘要。
public final void update(byte[] input, int offset, int len)
```

当然，我们还可以使用缓冲方式，方法如下所示：

```
// 使用指定的字节缓冲更新摘要。
public final void update(ByteBuffer input)
```

与MessageDigest类相同，更新摘要信息，其参数可以是更新一个字节、一个字节数组甚至是字节数组中的某一段偏移量，也可以是字节缓冲对象。

这些方法的调用顺序不受限制，在向摘要中增加所需数据时可以多次调用。

在完成摘要更新后，我们可以通过以下方法完成摘要操作：

```
// 完成摘要操作
public final byte[] doFinal()
/* 使用指定的字节数组对摘要进行最后更新，然后完成摘要计算，返回消息摘要字节数组。*/
public final byte[] doFinal(byte[] input)
// 完成摘要操作，按指定的偏移量将摘要信息保存在字节数组中。
public final void doFinal(byte[] output, int outOffset)
```

与MessageDigest类相同，Mac类也有重置方法：

```
// 重置摘要以供再次使用。
public final void reset()
```

执行该重置方法等同于创建一个新的Mac实例化对象。

除了上述方法外，还常用到以下几个方法：

```
/* 返回以字节为单位的摘要长度，如果提供者不支持此操作并且实现是不可复制的，则返回0。*/
public final int getMacLength()
// 返回算法名称，如HmacMD5。
public final String getAlgorithm()
```

更多资源请访问稀酷客(www.ckook.com)

```
// 返回此信息摘要对象的提供者。
public final Provider getProvider()
```

❑ 实现示例

安全消息摘要算法实现也较为简单，如代码清单3-17所示。

代码清单3-17 HmacMD5算法摘要处理

```
// 待做安全消息摘要的原始信息
byte[] input = "MAC".getBytes();
// 初始化KeyGenerator对象，使用HmacMD5算法
KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacMD5");
// 构建SecretKey对象
SecretKey secretKey = keyGenerator.generateKey();
// 构建Mac对象
Mac mac = Mac.getInstance(secretKey.getAlgorithm());
// 初始化Mac对象
mac.init(secretKey);
// 获得经过安全消息摘要后的信息
byte[] output = mac.doFinal(input);
```

3.3.2 KeyGenerator

KeyGenerator类与KeyPairGenerator类相似，KeyGenerator类用来生成秘密密钥，我们称它为秘密密钥生成器。

```
// 生成用于对称加密算法的秘密密钥，并提供相关信息。
public class KeyGenerator
extends Object
```

Java 6版本中提供了Blowfish、AES、DES和DESede等多种对称加密算法实现，以及HmacMD5、HmacSHA1和HmacSHA256等多种安全消息摘要算法实现。

关于Java 6版本中提供的算法信息，请参见附录。

❑ 方法详述

与KeyPairGenerator类相似，KeyGenerator类通过如下方法获得实例化对象：

```
// 返回生成指定算法的秘密密钥的KeyGenerator对象。
public static final KeyGenerator getInstance(String algorithm)
```

另一种方式就是指定算法名称的同时指定该算法的提供者，方法如下所示：

```
// 返回生成指定算法的秘密密钥的KeyGenerator对象。
public static final KeyGenerator getInstance(String algorithm, Provider provider)
// 返回生成指定算法的秘密密钥的KeyGenerator对象。
public static final KeyGenerator getInstance(String algorithm, String provider)
```

KeyGenerator对象可重复使用，也就是说，在生成密钥后，可以重复使用同一个KeyGenerator对象来生成更多的密钥。

更多资源请访问稀酷客(www.ckook.com)

生成密钥的方式有两种：与算法无关的方式和特定于算法的方式。这一点与KeyPairGenerator类生成密钥方式相类似。

两者之间的唯一不同是对象的初始化：

• 与算法无关的初始化

所有密钥生成器都具有密钥大小和随机源的概念。KeyGenerator类中有一个init()方法，它带有这两个通用共享类型的参数。还有一个只带有keysize参数的init()方法，它使用具有最高优先级的已安装提供者的SecureRandom实现作为随机源（如果已安装的提供者都不提供SecureRandom实现，则使用系统提供的随机源）。KeyGenerator类还提供一个只带随机源参数的init()方法。

因为调用上述与算法无关的init()方法时未指定其他参数，所以由提供者决定如何处理要与每个密钥关联的特定于算法的参数（如果有）。

• 特定于算法的初始化

在已经存在特定于算法的参数集的情况下，有两个带AlgorithmParameterSpec参数的 init()方法。其中一个方法还有一个SecureRandom参数，而另一个方法将具有高优先级的已安装提供者的SecureRandom实现作为随机源（如果安装的提供者都不提供SecureRandom实现，则使用系统提供的随机源）。

如果客户端没有显式地初始化KeyGenerator（通过调用init()方法），那么每个提供者都必须提供（并记录）默认初始化。

与算法无关的初始化方法如下：

```
// 初始化此KeyGenerator。
public final void init(SecureRandom random)
// 初始化此KeyGenerator，使其具有确定的密钥大小。
public final void init(int keysize)
// 使用用户提供的随机源初始化此KeyGenerator，使其具有确定的密钥大小。
public final void init(int keysize, SecureRandom random)
```

特定于算法的初始化方法如下：

```
// 用指定参数集初始化此KeyGenerator。
public final void init(AlgorithmParameterSpec params)
// 用指定参数集和用户提供的随机源初始化此KeyGenerator。
public final void init(AlgorithmParameterSpec params, SecureRandom random)
```

完成初始化操作后，我们就可以通过以下方法获得秘密密钥：

```
// 生成一个SecretKey对象。
public final SecretKey generateKey()
```

与其他引擎类一样，KeyGenerator类提供如下两种方法：

```
// 返回此秘密密钥生成器对象的算法名称。
public final String getAlgorithm()
// 返回此秘密密钥生成器对象的提供者。
public final Provider getProvider()
```

更多资源请访问稀酷客(www.ckook.com)

❑ 实现示例

```
// 实例化KeyGenerator对象，并指定HmacMD5算法。
KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacMD5");
// 生成SecretKey对象。
SecretKey secretKey = keyGenerator.generateKey();
```

3.3.3 KeyAgreement

KeyAgreement类提供密钥协定协议的功能，它同样是一个引擎类。我们称它为密钥协定，将在DH算法实现中使用到它。

```
// 此类提供密钥协定（或密钥交换）协议的功能。
public class KeyAgreement
extends Object
```

❑ 方法详述

与我们所熟悉的其他引擎类一样，KeyAgreement类需要通过getInstance()工厂方法来获得实例化对象：

```
// 返回实现指定密钥协定算法的KeyAgreement对象。
public static KeyAgreement getInstance(String algorithm)
// 返回实现指定密钥协定算法的KeyAgreement对象。
public static KeyAgreement getInstance(String algorithm, Provider provider)
// 返回实现指定密钥协定算法的KeyAgreement对象。
public static KeyAgreement getInstance(String algorithm, String provider)
```

算法生成器共有两种初始化方式：与算法无关的方式或特定于算法的方式。

获得实例化对象后，需要执行以下初始化方法：

```
/* 用给定密钥初始化此KeyAgreement，给定密钥需要包含此KeyAgreement所需的所有算法参数。*/
public void init(Key key)
```

当然，我们也可以指定密钥的同时给出对应的算法参数，方法如下所示：

```
// 用给定密钥和算法参数集初始化此KeyAgreement。
public void init(Key key, AlgorithmParameterSpec params)
```

或者，基于上述方式，再加入安全随机数参数，方法如下所示：

```
// 用给定密钥、算法参数集和随机源初始化此KeyAgreement。
public void init(Key key, AlgorithmParameterSpec params, SecureRandom random)
```

除上述方式外，我们也可以仅使用密钥和安全随机数两个参数完成初始化操作，方法如下所示：

```
// 用给定密钥和随机源初始化此KeyAgreement。
public void init(Key key, SecureRandom random)
```

然后，我们需要调用如下方法执行计划：

```
/*  用给定密钥执行此KeyAgreement的下一个阶段，给定密钥是从此密钥协定所涉及的其他某个参与者那里
```

更多资源请访问稀酷客(www.ckook.com)

```
接收的。*/
public Key doPhase(Key key, boolean lastPhase)
```

最后，我们可以获得共享秘密密钥：

```
// 生成共享秘密密钥并在新的缓冲区中返回它。
public byte[] generateSecret()
// 生成共享秘密密钥，并将其放入缓冲区 sharedSecret，从 offset（包括）开始。
public int generateSecret(byte[] sharedSecret, int offset)
// 创建共享秘密密钥并将其作为指定算法的SecretKey对象。
public SecretKey generateSecret(String algorithm)
```

此外，KeyAgreement类还提供了以下常用方法：

```
// 返回此密钥协定对象的提供者。
public Provider getProvider()
// 返回此密钥协定对象的算法名称。
public String getAlgorithm()
```

❑ 实现示例

KeyPairGenerator的实现是离不开DH算法的，如代码清单3-18所示。

代码清单3-18 DH算法密钥对生成

```
// 实例化KeyPairGenerator对象，并指定DH算法。
KeyPairGenerator kpg = KeyPairGenerator.getInstance("DH");
// 生成KeyPair对象kp1
KeyPair kp1 = kpg.genKeyPair();
// 生成KeyPair对象kp2
KeyPair kp2 = kpg.genKeyPair();
// 实例化KeyAgreement对象
KeyAgreement keyAgree = KeyAgreement.getInstance(kpg.getAlgorithm());
// 初始化KeyAgreement对象
keyAgree.init(kp2.getPrivate());
// 执行计划
keyAgree.doPhase(kp1.getPublic(), true);
// 生成SecretKey对象
SecretKey secretKey = keyAgree.generateSecret("DES");
```

3.3.4 SecretKeyFactory

SecretKeyFactory类同样属于引擎类，与KeyFactory类相对应，它用于产生秘密密钥，我们称它为秘密密钥工厂。

```
// 此类表示秘密密钥的工厂。
public class SecretKeyFactory
extends Object
```

❑ 方法详述

既然SecretKeyFactory类也是一个引擎类，那同样需要通过getInstance()工厂方法来实例化

更多资源请访问稀酷客(www.ckook.com)

对象。

我们可以通过制定算法名称的方式获得秘密密钥实例化对象，方法如下所示：

```
// 返回转换指定算法的秘密密钥的SecretKeyFactory对象。
public final static SecretKeyFactory getInstance(String algorithm)
```

或者，指定算法名称的同时制定该算法的提供者，方法如下所示：

```
// 返回转换指定算法的秘密密钥的SecretKeyFactory对象。
public final static SecretKeyFactory getInstance(String algorithm, Provider provider)
// 返回转换指定算法的秘密密钥的SecretKeyFactory对象。
public final static SecretKeyFactory getInstance(String algorithm, String provider)
```

算法生成器共有两种初始化方式：与算法无关的方式或特定于算法的方式。

得到SecretKeyFactory实例化对象后，我们就可以通过以下方法来生成秘密密钥：

```
// 根据提供的密钥规范（密钥材料）生成SecretKey对象。
public final SecretKey generateSecret(KeySpec keySpec)
```

此外，还可以使用如下方法转换秘密密钥：

```
/* 将一个密钥对象（其提供者未知或可能不受信任）转换为此SecretKeyFactory的相应密钥对象。*/
public final SecretKey translateKey(SecretKey key)
```

此外，可以使用如下方法获得秘密密钥的密钥规范：

```
// 以输入参数的格式返回给定密钥对象的规范（密钥材料）。
public final KeySpec getKeySpec(SecretKey key, Class keySpec)
```

与其他引擎类一样，SecretKeyFactory类提供以下两种方法：

```
// 返回此秘密密钥工厂对象的提供者。
public final Provider getProvider()
// 返回此秘密密钥工厂对象的算法名称。
public final String getAlgorithm()
```

❑ 实现示例

我们通过以下代码获得秘密密钥的密钥编码字节数组：

```
// 实例化KeyGenerator对象，并指定DES算法。
KeyGenerator keyGenerator = KeyGenerator.getInstance("DES");
// 生成SecretKey对象
SecretKey secretKey = keyGenerator.generateKey();
// 获得秘密密钥的密钥编码字节数组。
byte[] key = secretKey.getEncoded();
```

得到上述密钥编码字节数组后，我们就可以还原其秘密密钥对象：

```
// 由获得的密钥编码字节数组构建DESKeySpec对象。
DESKeySpec dks = new DESKeySpec(key);
// 实例化SecretKeyFactory对象
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES");
// 生成SecretKey对象
SecretKey secretKey = keyFactory.generateSecret(dks);
```

更多资源请访问稀酷客(www.ckook.com)

至此，我们完成了Java平台中有关密钥构建Java API学习。在3.3.5节中，我们将详述如何使用Cipher类实现加密与解密算法。

3.3.5 Cipher

Cipher类为加密和解密提供密码功能。它构成了Java Cryptographic Extension（JCE）框架的核心。在本章的上述内容中，只完成了密钥的处理，并未完成加密与解密的操作。这些核心操作需要通过Cipher类来实现。

```
// 此类为加密和解密提供密码功能。
public class Cipher
extends Object
```

Cipher类是一个引擎类，它需要通过getInstance()工厂方法来实例化对象。

我们可以通过指定转换模式的方式获得实例化对象，方法如下所示：

```
// 返回实现指定转换的 Cipher对象。
public static Cipher getInstance(String transformation)
```

也可以在制定转换模式的同时制定该转换模式的提供者，方法如下所示：

```
// 返回实现指定转换的 Cipher对象。
public static Cipher getInstance(String transformation, Provider provider)
// 返回实现指定转换的 Cipher对象。
public static Cipher getInstance(String transformation, String provider)
```

注意这里的参数String transformation，通过如下代码示例：

```
Cipher c = Cipher.getInstance("DES");
```

上述实例化操作是一种最为简单的实现，并没有考虑DES分组算法的工作模式和填充模式，可通过以下方式对其设定：

```
Cipher c = Cipher.getInstance("DES/CBC/PKCS5Padding");
```

参数String transformation的格式是“算法/工作模式/填充模式”，不同的算法支持不同的工作模式以及填充模式。具体内容请参见附录。

在对Cipher对象进行初始化前，我们先来认识如下常量：

```
// 用于将Cipher初始化为解密模式的常量。
public final static int DECRYPT_MODE
// 用于将Cipher初始化为加密模式的常量。
public final static int ENCRYPT_MODE
```

通过这两个常量来完成用于加密或是解密操作的初始化，可以使用如下这个最为简单也是最为常用的方法：

```
// 用密钥初始化此 Cipher对象。
public final void init(int opmode, Key key)
```

或者使用算法参数规范或算法参数来完成初始化：

更多资源请访问稀酷客(www.ckook.com)

```
// 用密钥和一组算法参数初始化此Cipher对象。
public final void init(int opmode, Key key, AlgorithmParameters params)
// 用密钥和一组算法参数初始化此Cipher对象。
public final void init(int opmode, Key key, AlgorithmParameterSpec params)
```

以下三个方法加入了SecureRandom参数：

```
// 用一个密钥、一组算法参数和一个随机源初始化此Cipher对象。
public final void init(int opmode, Key key, AlgorithmParameterSpec params,
SecureRandom random)
// 用一个密钥、一组算法参数和一个随机源初始化此Cipher对象。
public final void init(int opmode, Key key, AlgorithmParameters params,
SecureRandom random)
// 用密钥和随机源初始化此Cipher对象。
public final void init(int opmode, Key key, SecureRandom random)
```

通过以下方法可借助于证书，获取其公钥来完成加密和解密操作：

```
// 用取自给定证书的公钥初始化此Cipher对象。
public final void init(int opmode, Certificate certificate)
// 用取自给定证书的公钥和随机源初始化此Cipher对象。
public final void init(int opmode, Certificate certificate, SecureRandom random)
```

如果需要多次更新待加密（解密）的数据可使用如下方法。

最为常用的是通过输入给定的字节数组完成更新：

```
/* 继续多部分加密或解密操作（具体取决于此Cipher对象的初始化方式），以处理其他数据部分。*/
public final byte[] update(byte[] input)
```

或者通过偏移量的方式完成更新，方法如下所示：

```
/* 继续多部分加密或解密操作（具体取决于此Cipher对象的初始化方式），以处理其他数据部分。*/
public final byte[] update(byte[] input, int inputOffset, int inputLen)
```

另外一种方式就是将更新结果输出至参数中，方法如下所示：

```
/* 继续多部分加密或解密操作（具体取决于此Cipher对象的初始化方式），以处理其他数据部分。*/
public final int update(byte[] input, int inputOffset, int inputLen, byte[] output)
/* 继续多部分加密或解密操作（具体取决于此Cipher对象的初始化方式），以处理其他数据部分。*/
public final int update(byte[] input, int inputOffset, int inputLen, byte[]
output, int outputOffset)
```

当然，我们也可以使用如下缓冲方式：

```
/* 继续多部分加密或解密操作（具体取决于此Cipher对象的初始化方式），以处理其他数据部分。*/
public final int update(ByteBuffer input, ByteBuffer output)
```

完成上述数据更新后，直接执行如下方法：

```
// 结束多部分加密或解密操作（具体取决于此Cipher对象的初始化方式）。
public final byte[] doFinal()
```

如果，加密（解密）操作不需要多次更新数据可以直接执行如下方法：

```
// 按单部分操作加密或解密数据，或者结束一个多部分操作。
```

更多资源请访问稀酷客(www.ckook.com)

```
public final byte[] doFinal(byte[] input)
```

或按以下偏移量的方式完成操作：

```
// 按单部分操作加密或解密数据，或者结束一个多部分操作。
public final byte[] doFinal(byte[] input, int inputOffset, int inputLen)
```

以下方式将操作后的结果存于给定的参数中，与上述方法大同小异：

```
// 结束多部分加密或解密操作（具体取决于此 Cipher 的初始化方式）。
public final int doFinal(byte[] output, int outputOffset)
```

与上述方法不同的是，以下方法可用于多部分操作，并将操作结果存于给定参数中：

```
// 按单部分操作加密或解密数据，或者结束一个多部分操作。
public final int doFinal(byte[] input, int inputOffset, int inputLen, byte[] output)
// 按单部分操作加密或解密数据，或者结束一个多部分操作。
public final int doFinal(byte[] input, int inputOffset, int inputLen, byte[]
output, int outputOffset)
```

以下方法提供了一种基于缓冲的处理方式：

```
// 按单部分操作加密或解密数据，或者结束一个多部分操作。
public final int doFinal(ByteBuffer input, ByteBuffer output)
```

除了完成数据的加密与解密，Cipher类还提供了对密钥的包装与解包。

我们先来了解一下与密钥包装有关的常量：

```
// 用于将Cipher对象初始化为密钥包装模式的常量。
public final static int WRAP_MODE
```

这一常量需要在进行Cipher对象初始化时使用，给出如下示例代码：

```
cipher.init(Cipher.WRAP_MODE, secretKey); //初始化
```

在此之后我们就可以执行包装操作，可使用如下方法：

```
// 包装密钥
public final byte[] wrap(Key key)
```

解包操作需要如下常量执行初始化：

```
// 用于将Cipher初始化为密钥解包模式的常量。
public final static int UNWRAP_MODE
```

这个常量同样需要在初始化中执行，给出如下示例代码：

```
cipher.init(Cipher.UNWRAP _MODE, secretKey); //初始化
```

在此之后才能执行解包操作。

我们先来看一下解包方法：

```
// 解包一个以前包装的密钥。
public final Key unwrap(byte[] wrappedKey, String wrappedKeyAlgorithm, int wrappedKeyType)
```

上述方法中的参数int wrappedKeyType需要使用如下常量：

更多资源请访问稀酷客(www.ckook.com)

```
// 用于表示要解包的密钥为"私钥"的常量。
public final static int PRIVATE_KEY
// 用于表示要解包的密钥为"公钥"的常量。
public final static int PUBLIC_KEY
// 用于表示要解包的密钥为"秘密密钥"的常量。
public final static int SECRET_KEY
```

在执行包装操作时使用的是私钥就使用私钥常量，依此对应。

如果读者对第2章中有关分组加密工作模式的内容还有印象，应该记得文中曾提到过初始化向量。我们可以通过如下方法获得：

```
// 返回新缓冲区中的初始化向量 (IV)。
public final byte[] getIV()
```

通常，我们有必要通过如下方法来获悉当前转换模式所支持的密钥长度，方法如下所示：

```
// 根据所安装的 JCE 仲裁策略文件，返回指定转换的最大密钥长度。
public final static int getMaxAllowedKeyLength(String transformation)
```

分组加密中，每一组都有固定的长度，也称为块，以下方法可以获得相应的块大小：

```
// 返回块的大小（以字节为单位）。
public final int getBlockSize()
```

以下方法获得输出缓冲区字节长度：

```
/* 根据给定的输入长度 inputLen（以字节为单位），返回保存下一个 update 或 doFinal 操作结果所
需的输出缓冲区长度（以字节为单位）。*/
public final int getOutputSize(int inputLen)
```

我们也可以通过如下方法获得该Cipher对象的算法参数相关信息：

```
/* 根据仲裁策略文件，返回包含最大 Cipher 参数值的 AlgorithmParameterSpec 对象。*/
public final static AlgorithmParameterSpec getMaxAllowedParameterSpec(String transformation)
// 返回此 Cipher 使用的参数。
public final AlgorithmParameters getParameters()
```

此外，Cipher类还提供以下方法：

```
// 返回此 Cipher 使用的豁免 (exemption) 机制对象。
public final ExemptionMechanism getExemptionMechanism()
```

Cipher类作为一个引擎类，同样提供如下方法：

```
// 返回此 Cipher 对象的提供者。
public final Provider getProvider()
// 返回此 Cipher 对象的算法名称。
public final String getAlgorithm()
```

NullCipher和Cipher是什么关系？

我们会看到在API文档中有一个NullCipher类，它用来验证程序的有效性，并不提供具体的加密和解密实现。在验证程序的时候将会用到它。

更多资源请访问稀酷客(www.ckook.com)

❑ 实现示例

可通过如下代码完成密钥的包装和解包操作：

```
// 实例化KeyGenerator对象，并指定DES算法。
KeyGenerator keyGenerator = KeyGenerator.getInstance("DES");
// 生成SecretKey对象
SecretKey secretKey = keyGenerator.generateKey();
// 实例化Cipher对象
Cipher cipher = Cipher.getInstance("DES");
```

接下来执行包装操作：

```
// 初始化Cipher对象，用于包装。
cipher.init(Cipher.WRAP_MODE, secretKey);
// 包装秘密密钥
byte[] k = cipher.wrap(Key key);
```

得到字节数组k后，可以将其传递给需要解包的一方。

省去上述实例化操作用以下代码示例：

```
// 初始化Cipher对象，用于解包。
cipher.init(Cipher.UNWRAP_MODE, secretKey);
// 解包秘密密钥
Key key = cipher.unwrap(k, "DES", Cipher.SECRET_KEY);
```

如果要做加密操作可按参考如下代码：

```
// 初始化Cipher对象，用于加密操作。
cipher.init(Cipher.ENCRYPT_MODE, secretKey);
// 加密
byte[] input = cipher.doFinal("DES DATA".getBytes());
```

解密操作与之相对应：

```
// 初始化Cipher对象，用于解密操作。
cipher.init(Cipher.DECRYPT_MODE, secretKey);
// 解密
byte[] output = cipher.doFinal(input);
```

3.3.6 CipherInputStream

CipherInputStream和CipherOutputStream同属Cipher类的扩展，统称为密钥流。按流的输入和输出方式分为密钥输入流和密钥输出流。

```
// 提供密钥输入流
public class CipherInputStream
extends FilterInputStream
```

❑ 方法详述

可通过如下构造方法构造实例化对象：

更多资源请访问稀酷客(www.ckook.com)

```
// 根据 InputStream 和 Cipher 构造 CipherInputStream对象。
public CipherInputStream(InputStream is, Cipher c)
```

CipherInputStream类覆盖了FilterInputStream类的以下方法。

以下是输入流的读操作，与一般FilterInputStream的子类别无二致：

```
// 从该输入流读取下一数据字节。
public int read()
// 从该输入流将 b.length 个数据字节读入到字节数组中。
public int read(byte[] b)
// 从该输入流将 len 个字节数据读入到字节数组中。
public int read(byte[] b, int off, int len)
```

通常，我们可以通过如下方法获知是否还有可读入内容：

```
// 返回不发生阻塞地从此输入流读取的字节数。
public int available()
```

或者，直接跳过某些内容，方法如下所示：

```
// 跳过不发生阻塞地从该输入流读取的字节中的 n 个字节的输入。
public long skip(long n)
```

我们可以通过如下方法验证该输入流是否支持标记和重置操作：

```
// 测试该输入流是否支持 mark 和 reset 方法以及哪一种方法确实不受支持。
public boolean markSupported()
```

完成操作后，一定要执行关闭流操作，方法如下所示：

```
// 关闭该输入流并释放任何与该流关联的系统资源。
public void close()
```

❑ 实现示例

我们通过如下代码来展示如何使用密钥输入流解密文件中的数据。

首先，构建Cipher实例化对象：

```
// 实例化KeyGenerator对象，指定DES算法。
KeyGenerator kg = KeyGenerator.getInstance("DES");
// 生成SecretKey对象
SecretKey secretKey = kg.generateKey();
// 实例化Cipher对象
Cipher cipher = Cipher.getInstance("DES");
```

接着我们从文件中读入数据，然后进行解密操作：

```
// 初始化Cipher对象，用于解密操作。
cipher.init(Cipher.DECRYPT_MODE, secretKey);
// 实例化CipherInputStream对象。
CipherInputStream cis = new CipherInputStream(new FileInputStream(new
File("secret")), cipher);
// 使用DataInputStream对象包装CipherInputStream对象。
DataInputStream dis = new DataInputStream(cis);
```

更多资源请访问稀酷客(www.ckook.com)

```
// 读出解密后的数据
String output = dis.readUTF();
// 关闭流
dis.close();
```

在这里，我们就能获得解密后的数据了。这个加密数据如何写进文件呢？将在后面展示。

3.3.7 CipherOutputStream

CipherOutputStream类与CipherInputStream类相似，称为密钥输出流。

```
// 提供密钥输出流
public class CipherOutputStream
extends FilterOutputStream
```

❑ 方法详述

可通过如下构造方法构造实例化对象：

```
// 通过 OutputStream 和 Cipher 构造 CipherOutputStream对象。
public CipherOutputStream(OutputStream os, Cipher c)
```

CipherOutputStream类覆盖了FilterOutputStream类的以下方法。

写操作很简单，与一般FilterOutputStream类的子类差别不大，方法如下所示：

```
// 从指定的字节数组中将 b.length 个字节写入此输出流。
public void write(byte[] b)
// 将指定的字节数组中从 off 偏移量开始的 len 个字节写入此输出流。
public void write(byte[] b, int off, int len)
// 将指定的字节写入此输出流。
public void write(int b)
```

完成操作后，一定要对输出流做清空和关闭操作，方法如下所示：

```
// 强制写出已由封装的密码对象处理的任何缓存输出字节来刷新此输出流。
public void flush()
// 关闭此输出流并释放任何与此流关联的系统资源。
public void close()
```

❑ 实现示例

接上3.3.6节内容，我们通过如下代码清单3-19完成文件数据的加密。

代码清单3-19 密钥输出流加密操作

```
// 初始化Cipher对象，用于加密操作。
cipher.init(Cipher.ENCRYPT_MODE, secretKey);
// 待加密的原始数据。
String input = "1234567890";
// 实例化CipherOutputStream对象。
CipherOutputStream cos = new CipherOutputStream(new FileOutputStream(new
File("secret")), cipher);
// 使用DataOuputStream对象包装CipherOutputStream对象。
```

更多资源请访问稀酷客(www.ckook.com)

```
DataOutputStream dos = new DataOutputStream(cos);
// 向输出流写待加密的数据。
dos.writeUTF(input);
// 清空流
dos.flush();
// 关闭流
dos.close();
```

3.3.8 SealedObject

SealedObject类使程序员能够用加密算法创建对象并保护其机密性。

```
// 此类使程序员能够用加密算法创建对象并保护其机密性。
public class SealedObject
extends Object
implements Serializable
```

❑ 方法详述

通过以下方法可以构建一个实例化对象：

```
// 从序列化对象构造一个 SealedObject。
public SealedObject(Serializable object, Cipher c)
```

在给定任何Serializable对象的情况下，程序员可以序列化格式（即“深层复制”）封装原始对象的SealedObject，并使用DES一类的加密算法密封（加密）其序列化的内容，以保护其机密性。加密的内容以后可以解密（使用相应的算法和正确的解密密钥）和反序列化，并生成原始对象。

注意，该Cipher对象在应用于SealedObject之前必须使用正确的算法、密钥、填充方案等进行完全初始化。

已密封的原始对象可以用以下两种方式恢复：

• 使用采用Cipher对象的getObject()方法。

此方法需要一个完全初始化的Cipher对象，用极其相同的用来密封对象的算法、密钥、填充方案等进行初始化。

这样做的好处是解封密封对象的一方不需要知道解密密钥。例如，一方用所需的解密密钥初始化Cipher对象之后，它就会将Cipher对象移交给以后要解封密封对象的另一方。

• 使用采用Key对象的getObject()方法。

在此方法中，getObject()方法创建一个用于适当解密算法的Cipher对象，并用给定的解密密钥和存储在密封对象中的算法参数（如果有）对其进行初始化。

这样做的好处是解封此对象的一方不需要跟踪用来密封该对象的参数（如IV，初始化向量）。

使用采用Cipher对象的getObject()方法：

```
// 获取原始（封装的）对象。
public final Object getObject(Cipher c)
```

更多资源请访问稀酷客(www.ckook.com)

使用采用Key对象的getObject()方法：

```
// 获取原始（封装的）对象。
public final Object getObject(Key key)
// 获取原始（封装的）对象。
public final Object getObject(Key key, String provider)
```

此外，提供如下常用方法：

```
// 返回用于密封此对象的算法。
public final String getAlgorithm()
```

❑ 实现示例

我们通过代码清单3-20展示如何对对象加密。

代码清单3-20 对象加密处理

```
// 待加密的字符串对象。
String input = "SealedObject";
// 实例化KeyGenerator对象，使用DES算法。
KeyGenerator kg = KeyGenerator.getInstance("DES");
// 创建秘密密钥
SecretKey key = kg.generateKey();
// 实例化用于加密的Cipher对象cipher1。
Cipher cipher1 = Cipher.getInstance(key.getAlgorithm());
// 初始化
cipher1.init(Cipher.ENCRYPT_MODE, key);
// 构建SealedObject对象
SealedObject sealedObject = new SealedObject(input, cipher1);
// 实例化用于解密的Cipher对象cipher2。
Cipher cipher2 = Cipher.getInstance(key.getAlgorithm());
// 初始化
cipher2.init(Cipher.DECRYPT_MODE, key);
// 获得解密后的字符串对象。
String output = (String) sealedObject.getObject(cipher2);
```

3.4 java.security.spec包和javax.crypto.spec包详解

java.security.spec包和javax.crypto.spec包都提供了密钥规范和算法参数规范的类和接口。获得密钥规范后，我们将有机会还原密钥对象。本节将详述KeySpec接口及其实现。

3.4.1 KeySpec和AlgorithmParameterSpec

KeySpec和AlgorithmParameterSpec是java.security.spec包中两个较为重要的接口。这两个接口本身都是空接口，仅用于将所有参数规范分组，并为其提供类型安全。

1. KeySpec

此接口不包含任何方法或常量。它仅用于将所有密钥规范分组，并为其提供类型安全。所

更多资源请访问稀酷客(www.ckook.com)

有密钥规范都必须实现此接口。

```
// 组成加密密钥的密钥内容的（透明）规范。
public interface KeySpec
```

KeySpec的抽象实现类（EncodedKeySpec）构建了用于构建公钥规范和私钥规范的两个实现（X509EncodedKeySpec用于构建公钥规范，PKCS8EncodedKeySpec用于构建私钥规范）。

SecretKeySpec接口是KeySpec的实现类，用于构建秘密密钥规范。

2. AlgorithmParameterSpec

此接口不包含任何方法或常量。它仅用于将所有参数规范分组，并为其提供类型安全。所有参数规范都必须实现此接口。

```
// 加密参数的（透明）规范。
public interface AlgorithmParameterSpec
```

AlgorithmParameterSpec接口有很多的子接口和实现类，用于特定于算法的初始化。使用起来也很方便，只需要使用指定参数填充构造方法即可获得一个实例化对象。我们以DSAParameterSpec为例：

```
// 此类指定用于DSA算法的参数的集合。
public class DSAParameterSpec
extends Object
implements AlgorithmParameterSpec, DSAParams
```

通过以下方法获得实例化对象：

```
// 创建一个具有指定参数值的新的DSAParameterSpec。
public DSAParameterSpec(BigInteger p, BigInteger q, BigInteger g)
```

在上述参数中，p为素数、q为子素数、g为基数。通过给定的数学参数构造实例化对象。

获得对象后可通过如下方法获得其属性值：

```
// 返回基数g
publi c BigInteger getG()
// 返回素数p
public BigInteger getP()
// 返回子素数q
public BigInteger getQ()
```

此外，还有很多AlgorithmParameterSpec接口的实现，基本上与DSAParameterSpec类的使用方式相似，有兴趣的读者可以查看相应的API文档。

3.4.2 EncodedKeySpec

EncodedKeySpec类用编码格式来表示公钥和私钥，我们称它为编码密钥规范。编码密钥规范的实现子类很多，在这里不一一详述。本文将详细介绍最为常用的用于表示公钥和私钥的两个实现类。

X509EncodedKeySpec和PKCS8EncodedKeySpec两个类均为EncodedKeySpec的子类，

更多资源请访问稀酷客(www.ckook.com)

X509EncodedKeySpec类用于转换公钥编码密钥，PKCS8EncodedKeySpec类用于转换私钥编码密钥。

```
// 此类用编码格式表示公钥或私钥。
public abstract class EncodedKeySpec
extends Object
implements KeySpec
```

以下是该类的方法，我们将通过其两个重要的子类来详述如何使用：

```
// 根据给定的编码密钥创建一个新的编码密钥规范。
public EncodedKeySpec(byte[] encodedKey)
// 返回编码密钥。
public byte[] getEncoded()
// 返回与此密钥规范相关联的编码格式的名称。
public abstract  String getFormat()
```

1. X509EncodedKeySpec

X509EncodedKeySpec类继承EncodedKeySpec类，以编码格式来表示公钥。

X509EncodedKeySpec类使用X.509标准作为密钥规范管理的编码格式，该类的命名由此得来。

```
// 用编码格式表示公用。
public class X509EncodedKeySpec
extends EncodedKeySpec
```

❑ 方法详述

可通过如下方法实例化对象：

```
// 根据给定的编码密钥创建一个新的 X509EncodedKeySpec。
public X509EncodedKeySpec(byte[] encodedKey)
```

以下两种方法均依照X.509标准：

```
// 返回按照X.509标准进行编码的密钥的字节。
public byte[] getEncoded()
//返回与此密钥规范相关联的编码格式的名称。将得到字符串"X.509"。
public String getFormat()
```

❑ 实现示例

首先，我们通过如下代码获得密钥对：

```
// 实例化KeyPairGenerator对象，并指定DSA算法。
KeyPairGenerator keygen = KeyPairGenerator.getInstance("DSA");
// 初始化KeyPairGenerator对象。
keygen.initialize(1024);
// 生成KeyPair对象。
KeyPair keys = keygen.genKeyPair();
```

其次，我们获得公钥的密钥字节数组：

更多资源请访问稀酷客(www.ckook.com)

```
// 获得公钥密钥字节数组。
byte[] publicKeyBytes = keys.getPublic().getEncoded();
```

最后，我们将获得的公钥密钥字节数组再转换为公钥对象：

```
// 实例化X509EncodedKeySpec对象。
X509EncodedKeySpec keySpec = new X509EncodedKeySpec(publicKeyBytes);
// 实例化KeyFactory对象，并指定DSA算法。
KeyFactory keyFactory = KeyFactory.getInstance("DSA");
// 获得PublicKey对象。
PublicKey publicKey = keyFactory.generatePublic(keySpec);
```

私钥对象的转换与此大致相同。

2. PKCS8EncodedKeySpec

PKCS8EncodedKeySpec类继承EncodedKeySpec类，以编码格式来表示私钥。

PKCS8EncodedKeySpec类使用PKCS#8标准作为密钥规范管理的编码格式，该类的命名由此得来。

```
// 用编码格式来表示私钥。
public class PKCS8EncodedKeySpec
extends EncodedKeySpec
```

❑ 方法详述

可通过如下方法实例化对象：

```
// 根据给定的编码密钥创建一个新的 PKCS8EncodedKeySpec。
public PKCS8EncodedKeySpec(byte[] encodedKey)
```

以下两种方法均依照PKCS #8标准：

```
// 返回按照 PKCS#8标准编码的密钥字节。
public byte[] getEncoded()
// 返回与此密钥规范相关联的编码格式的名称。将得到字符串"PKCS#8"。
public String getFormat()
```

❑ 实现示例

下面根据前面介绍的实现示例来展示如何获得私钥对象。

通过获得的密钥对，我们来获得私钥密钥编码字节数组：

```
// 获得私钥密钥字节数组。
byte[] privateKeyBytes = keys.getPrivate().getEncoded();
```

最后，我们将获得私钥对象：

```
// 实例化PKCS8EncodedKeySpec对象。
PKCS8EncodedKeySpec keySpec = new PKCS8EncodedKeySpec(privateKeyBytes);
// 实例化KeyFactory对象，并指定DSA算法。
KeyFactory keyFactory = KeyFactory.getInstance("DSA");
// 获得PrivateKey对象。
PrivateKey privateKey = keyFactory.generatePrivate (keySpec);
```

更多资源请访问稀酷客(www.ckook.com)

X509EncodedKeySpec和PKCS8EncodedKeySpec两个类在加密解密环节中经常会用到。密钥很可能会以二进制方式存储于文件中，由程序来读取。这时候，就需要通过这两个类将文件中的字节数组读出转换为密钥对象。

3.4.3 SecretKeySpec

SecretKeySpec类是KeySpec接口的实现类，用于构建秘密密钥规范。可根据一个字节数组构造一个SecretKey，而无须通过一个（基于provider的）SecretKeyFactory。

```
// 此类以与provider无关的方式指定一个密钥。
public class SecretKeySpec
extends Object
implements KeySpec, SecretKey
```

此类仅对能表示为一个字节数组并且没有任何与之相关联的密钥参数的原始密钥有用，如DES或Triple DES密钥。

❑ 方法详述

可以通过下述方法构建一个实例化对象：

```
/* 根据给定的字节数组构造一个密钥，使用key中的始于且包含offset的前len个字节。*/
public SecretKeySpec(byte[] key, int offset, int len, String algorithm)
// 根据给定的字节数组构造一个密钥。
public SecretKeySpec(byte[] key, String algorithm)
```

SecretKeySpec还覆盖了以下方法：

```
// 测试给定对象与此对象的相等性。
public boolean equals(Object obj)
// 算此对象的散列码值。
public int hashCode()
```

此外，SecretKeySpec还提供了如下方法：

```
// 返回与此密钥相关联的算法的名称。
public String getAlgorithm()
// 返回此密钥的密钥内容。
public byte[] getEncoded()
// 返回此密钥编码格式的名称。
public String getFormat()
```

❑ 实现示例

我们先获得RC2算法的密钥字节数组：

```
// 实例化KeyGenerator对象，并指定RC2算法。
KeyGenerator kg = KeyGenerator.getInstance("RC2"));
// 生成SecretKey对象。
SecretKey secretKey = kg.generateKey();
// 获得密钥编码字节数组。
byte[] key = secretKey.getEncoded();
```

更多资源请访问稀酷客(www.ckook.com)

在得到密钥编码字节数组后，我们将通过如下方法还原秘密密钥对象：

```
// 实例化SecretKey对象。
SecretKey secretKey = new SecretKeySpec(key, "RC2");
```

3.4.4 DESKeySpec

DESKeySpec类与SecretKeySpec类都是提供秘密密钥规范的实现类。不同之处在于，DESKeySpec类指定了DES算法，SecretKeySpec类则是兼容所有对称加密算法。

DESKeySpec类有很多的同胞，例如，DESedeKeySpec类提供了三重DES加密算法的密钥规范；PBEKeySpec类提供了PBE算法的密钥规范。我们以DESKeySpec类为代表，对其同胞类做相应说明。

```
// 此类指定一个DES密钥。
public class DESKeySpec
extends Object
implements KeySpec
```

❑ 方法详述

可以通过如下构造方法获得实例化对象：

```
/* 创建一个DESKeySpec对象，使用key中的前8个字节作为DES密钥的密钥内容。*/
public DESKeySpec(byte[] key)
/* 创建一个DESKeySpec对象，使用key中始于且包含offset的前8个字节作为DES-EDE密钥的密钥内容。*/
public DESKeySpec(byte[] key, int offset)
```

在上述方法中的8个字节的定义就来源于这个常量定义，如下所示：

```
// 定义以字节为单位的DES密钥长度的常量（8）。
public static int DES_KEY_LEN
```

此外，DESKeySpec类还提供了如下方法：

```
// 返回DES密钥内容。
public byte[] getKey()
/* 确定给定的始于且包含offset的DES密钥内容是否是奇偶校验的（parity-adjusted）。*/
public static boolean isParityAdjusted(byte[] key, int offset)
// 确定给定的DES密钥内容是否是全弱或者半弱的。
public static boolean isWeak(byte[] key, int offset)
```

❑ 实现示例

我们修改3.4.3节的内容来演示如何使用DESKeySpec类还原密钥对象。

先获得DES算法的密钥字节数组：

```
// 实例化KeyGenerator对象，并指定DES算法。
KeyGenerator kg = KeyGenerator.getInstance("DES"));
// 生成SecretKey对象。
SecretKey secretKey = kg.generateKey();
// 获得密钥编码字节数组。
byte[] key = secretKey.getEncoded();
```

更多资源请访问稀酷客(www.ckook.com)

在得到密钥编码字节数组后，我们将通过如下方法还原秘密密钥对象：

```
// 指定DES算法，还原SecretKey对象。
SecretKey secretKey = new SecretKeySpec(key, "DES");
```

如果使用DESKeySpec该如何做呢？在得到了密钥编码之后，可通过如下代码来实现：

```
// 实例化DESKeySpec对象，获得DES秘密密钥规范。
DESKeySpec dks = new DESKeySpec(key);
// 实例化SecretKeyFactory对象，并指定DES算法。
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES");
// 获得SecretKey对象
SecretKey secretKey = keyFactory.generateSecret(dks);
```

如果是三重DES算法，该怎么做呢？除了原有指明为“DES”算法的位置替换为“DESede”外，只需要把DESKeySpec类换为DESedeKeySpec类即可。示例代码如下：

```
// 实例化，并指定DESede算法。
KeyGenerator kg = KeyGenerator.getInstance("DESede"));
// 生成SecretKey对象。
SecretKey secretKey = kg.generateKey();
// 获得密钥编码字节数组。
byte[] key = secretKey.getEncoded();
```

得到密钥编码之后，可通过如下代码来实现：

```
// 实例化DESedeKeySpec对象，获得DESede秘密密钥规范。
DESedeKeySpec dks = new DESedeKeySpec(key);
// 实例化SecretKeyFactory对象，并指定DESede算法。
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DESede");
// 获得SecretKey对象
SecretKey secretKey = keyFactory.generateSecret(dks);
```

其他对称秘密密钥规范的使用与上述代码实现大致相同，有兴趣的读者可以参考相应的API内容。

3.5 java.security.cert包详解

java.security.cert包提供用于解析和管理证书、证书撤销列表（CRL）和证书路径的类和接口。作为Java加密与解密实现的扩展。本节着重详述了Certificate类及其子类和CertificateFactory类的使用。作为补充，本节详述了有关证书撤销相关的CRL类及其子类和用于证书路径的类CertPath类的使用。对证书有兴趣的读者可以参考相应的Java API内容。

3.5.1 Certificate

Certificate类是一个用于管理证书的抽象类。证书有多种类型，如X.509、PGP和SDSI证书，并且它们以不同的方式存储和存储信息，但都可以通过继承Certificate类来实现它们。

更多资源请访问稀酷客(www.ckook.com)

```
// 管理各种身份证书的抽象类。
public abstract class Certificate
extends Object
implements Serializable
```

❑ 方法详述

Certificate类提供3种基本操作，要求子类实现：

```
// 返回此证书的编码形式。
public abstract byte[] getEncoded()
// 验证是否已使用与指定公钥相应的私钥签署了此证书。
public abstract void verify(PublicKey key)
// 验证是否已使用与指定公钥相应的私钥签署了此证书。
public abstract void verify(PublicKey key, String sigProvider)
// 从此证书中获取公钥。
public abstract PublicKey getPublicKey()
```

Certificate类还要求子类必须实现以下方法：

```
// 返回此证书的字符串表示形式。
public abstract String toString()
```

通过如下方法将得到该证书的类型：

```
// 返回此证书的类型。如X.509、PGP和SDSI。
public final String getType()
```

此外，Certificate类覆盖了以下方法：

```
// 根据此证书的编码形式返回该证书的散列码值。
public int hashCode()
// 比较此证书与指定对象的相等性。
public boolean equals(Object other)
```

Certificate类有一个抽象子类——X509Certificate类，我们将在后面几节内容中介绍它。

3.5.2 CertificateFactory

CertificateFactory类是一个引擎类，我们称它为证书工厂，可以通过它将证书导入程序中。

```
/*  此类定义了用于从相关的编码中生成证书、证书路径(CertPath)和证书撤销列表(CRL)对象的
CertificateFactory功能。*/
public class CertificateFactory
extends Object
```

❑ 方法详述

可通过以下getInstance()工厂方法获得实例化对象：

```
// 返回实现指定证书类型的CertificateFactory对象。
public final static CertificateFactory getInstance(String type)
// 返回指定证书类型的CertificateFactory对象。
public final static CertificateFactory getInstance(String type, Provider provider)
```

更多资源请访问稀酷客(www.ckook.com)

```
// 返回指定证书类型的CertificateFactory对象。
public final static CertificateFactory getInstance(String type, String provider)
```

可以通过CertificateFactory类生成证书：

```
// 生成一个证书对象，并使用从输入流inStream中读取的数据对它进行初始化。
public final Certificate generateCertificate(InputStream inStream)
// 返回从给定输入流inStream中读取的证书的集合视图（可能为空）。
public final Collection<? extends Certificate> generateCertificates(InputStream inStream)
```

也可以通过CertificateFactory类生成证书路径。

以下方法可以通过输入流获得证书路径：

```
/* 生成一个CertPath对象，并使用从InputStream inStream中读取的数据对它进行初始化。*/
public final CertPath generateCertPath(InputStream inStream)
/* 生成一个CertPath对象，并使用从InputStream inStream中读取的数据对它进行初始化。*/
public final CertPath generateCertPath(InputStream inStream, String encoding)
```

我们也可以通过如下方法生成一个正式路径：

```
/* 生成一个CertPath对象，并使用一个Certificate的List对它进行初始化。*/
public final CertPath generateCertPath(List<? extends Certificate>certificates)
```

还可以通过CertificateFactory类生成证书撤销列表：

```
/* 生成一个证书撤销列表(CRL)对象，并使用从输入流inStream中读取的数据对它进行初始化。*/
public final CRL generateCRL(InputStream inStream)
/* 返回从给定输入流inStream中读取的CRL的集合视图（可能为空）。*/
public final Collection<? extends CRL> generateCRLs(InputStream inStream)
```

此外，CertificateFactory类还提供了以下方法：

```
/* 返回此CertificateFactory支持的CertPath编码的迭代器，默认编码方式优先。*/
public final Iterator<String> getCertPathEncodings()
// 返回此CertificateFactory的提供者。
public final Provider getProvider()
// 返回与此CertificateFactory相关联的证书类型的名称。
public final String getType()
```

❑ 实现示例

如果我们已知待载入证书的类型，就可通过代码清单3-21获得证书对象。

代码清单3-21 加载证书

```
// 实例化，并指明证书类型为"X.509"。
CertificateFactory certificateFactory = CertificateFactory.getInstance("X.509");
// 获得证书输入流
FileInputStream in = new FileInputStream("D:\\x.keystore");
// 获得证书
Certificate certificate = certificateFactory.generateCertificate(in);
// 关闭流
in.close();
```

更多资源请访问稀酷客(www.ckook.com)

3.5.3 X509Certificate

X509Certificate类是Certificate类的子类，它同样是一个抽象类。

```
// X.509证书的抽象类。此类提供了一种访问X.509证书所有属性的标准方式。
public abstract class X509Certificate
extends Certificate
implements X509Extension
```

❑ 方法详述

得到X.509类型的证书对象后，我们最先要做的事情就是校验证书是否有效。

证书的有效期是一个区间范围，也就是起止时间，可用以下两个方法校验：

```
// 获取证书有效期的notAfter日期。
public abstract Date getNotAfter()
// 获取证书有效期的 notBefore日期。
public abstract Date getNotBefore()
```

除了上述两种方法，还可以校验给定日期是否处于证书的有效期内：

```
// 检查给定的日期是否处于证书的有效期内。
public abstract void checkValidity(Date date)
```

下述这个方法就更为简单了，它将校验当前时间是否处于证书的有效期内：

```
// 检查证书目前是否有效。
public abstract void checkValidity()
```

除了上述校验功能外，还可通过以下方法获得证书的相应属性。

我们可以通过如下方法获得一些简单的证书基本信息：

```
// 获取证书的version（版本号）值。如1、2或3。
public abstract int getVersion()
// 获取证书的serialNumber值。
public abstract BigInteger getSerialNumber()
/* 从关键BasicConstraints扩展(OID = 2.5.29.19)中获取证书的限制路径长度。*/
public abstract int getBasicConstraints()
```

以下方法可以获得证书中KeyUsage的相关信息：

```
/* 获取一个表示KeyUsage扩展(OID = 2.5.29.15)的各个位的boolean数组。*/
public abstract boolean[] getKeyUsage()
/* 获取一个不可修改的String列表，表示已扩展的密钥使用扩展(OID = 2.5.29.37)中
ExtKeyUsageSyntax字段的对象标识符（OBJECT IDENTIFIER）。*/
public List<String> getExtendedKeyUsage()
```

我们可以通过如下方法获得证书的发布者的相关信息：

```
/* 从IssuerAltName扩展(OID = 2.5.29.18)中获取一个发布方替换名称的不可变集合。*/
public Collection<List<?>> getIssuerAlternativeNames()
// 获取证书的issuerUniqueID值。
public abstract boolean[] getIssuerUniqueID()
// 以X500Principal的形式返回证书的发布方（发布方标识名）值。
```

更多资源请访问稀酷客(www.ckook.com)

```
public X500Principal getIssuerX500Principal()
```

以下方法可以获得证书主体的一些相关信息：

```
/* 从SubjectAltName扩展(OID = 2.5.29.17)中获取一个主体替换名称的不可变集合。*/
public Collection<List<?>> getSubjectAlternativeNames()
// 获取证书的subjectUniqueID值。
public abstract boolean[] getSubjectUniqueID()
// 以X500Principal的形式返回证书的主体（主体标识名）值。
public X500Principal getSubjectX500Principal()
```

我们也可以获得证书的DER编码的二进制信息，方法如下所示：

```
// 从此证书中获取以DER编码的证书信息，即tbsCertificate。
public abstract byte[] getTBSCertificate()
```

从程序设计的角度来讲，我们更常用到以下这些方法：

```
// 获取证书签名算法的签名算法名。
public abstract String getSigAlgName()
// 获取证书的签名算法OID字符串。
public abstract String getSigAlgOID()
// 从此证书的签名算法中获取DER编码形式的签名算法参数。
public abstract byte[] getSigAlgParams()
```

在某些时候，我们更关注证书的签名值，方法如下所示：

```
// 获取证书的signature值（原始签名位）。
public abstract byte[] getSignature()
```

❑ 实现示例

我们通过以下代码清单3-22来展示如何通过密钥库获得证书。

代码清单3-22 获得证书签名

```
// 加载密钥库文件
FileInputStream is = new FileInputStream("D:\\x.keystore");
// 实例化KeyStore对象
KeyStore ks = KeyStore.getInstance("JKS");
// 加载密钥库
ks.load(is, "password".toCharArray());
// 关闭文件输入流
is.close();
// 获得X.509类型证书
X509Certificate x509Certificate = (X509Certificate) ks.getCertificate("alias");
// 通过证书标明的签名算法构建Signature对象。
Signature signature = Signature.getInstance(x509Certificate.getSigAlgName());
```

3.5.4 CRL

证书可能会由于各种原因失效，如由于申请证书的请求有问题，或者用户使用该证书做了非法操作。这时，证书将立即被置为无效。将证书置为无效的结果就是产生CRL（证书撤销列

更多资源请访问稀酷客(www.ckook.com)

表)。CA负责发布CRL，其中列出了该CA已经撤销的证书。验证证书时，首先需要查询此列表，然后再考虑接受证书的合法性。

CRL类作为证书抽象列表的抽象类，可通过扩展该抽象类定义专门的CRL类型。

```
// 此类是具有不同格式但很常用的证书撤销列表(CRL)的抽象。
public abstract class CRL
extends Object
```

❑ 方法详述

CRL类提供了获取CRL类型的方法：

```
// 返回此CRL的类型
public String getType()
```

此外，要求其子类必须实现以下方法：

```
/* 检查给定的证书是否在此CRL中。如果给定的证书在此CRL中，则返回true，否则返回false。*/
public abstract boolean isRevoked(Certificate cert)
// 返回此CRL的字符串表示形式
public abstract String toString()
```

其中，isRevoked()方法是我们最为常用的方法。

❑ 实现示例

CRL类的实例可通过代码清单3-23方式获得。

代码清单3-23 获得证书撤销列表

```
// 实例化，并指明证书类型为"X.509"。
CertificateFactory certificateFactory = CertificateFactory.getInstance("X.509");
// 获得证书输入流
FileInputStream in = new FileInputStream("D:\\x.keystore");
// 获得证书撤销列表
CRL crl = certificateFactory.generateCRL(in);
// 关闭流
in.close();
```

3.5.5 X509CRLEntry

X509CRLEntry类可用于撤销证书。

```
// 用于CRL（证书撤销列表）中已撤销证书的抽象类。
public abstract class X509CRLEntry
extends Object
implements X509Extension
```

❑ 方法详述

X509CRLEntry类实现了以下方法：

```
// 比较此CRL项与给定对象的相等性。
public boolean equals(Object other)
```

更多资源请访问稀酷客(www.ckook.com)

```
// 根据此CRL项的编码形式返回该CRL项的散列码值。
public int hashCode()
```

此外，要求子类实现如下方法。

以下方法可获得证书撤销列表实体的DER编码二进制信息：

```
// 返回此CRL Entry的ASN.1 DER编码形式，即内部序列值。
public abstract byte[] getEncoded()
```

我们可能最为关心的是下面几个方法：

```
// 获取此X509CRLEntry的撤销日期revocationDate。
public abstract Date getRevocationDate()
// 获取此X509CRLEntry的序列号userCertificate。
public abstract BigInteger getSerialNumber()
// 如果此CRL项有扩展，则返回true。
public abstract boolean hasExtensions()
```

子类还必须覆盖Object的下述方法：

```
// 返回此CRL项的字符串表示形式。
public abstract String toString()
```

在Java 5以后，X509CRLEntry加入了如下方法，但自身并无实现，返回值为null：

```
// 获取此项所描述的X509Certificate的发布方。
public X500Principal getCertificateIssuer()
```

3.5.6 X509CRL

X509CRL类作为CRL类的子类，已标明了类型为X.509的CRL。

```
/* X.509证书撤销列表(CRL)的抽象类。CRL是标识已撤销证书的时间戳列表。它由证书颁发机构(CA)签署
并且可在公共存储库中随意使用。*/
public abstract class X509CRL
extends CRL
implements X509Extension
```

❑ 方法详述

X509CRL类覆盖了以下两个方法：

```
// 比较此CRL与给定对象的相等性。
public boolean equals(Object other)
// 根据此CRL的编码形式返回该CRL的散列码值。
public int hashCode()
```

在获得X509CRL实例化对象后，我们可以通过以下方法获得相应属性。

我们可以通过如下方法获得版本信息：

```
// 获取CRL的version（版本号）值。
public abstract int getVersion()
```

可以通过如下方法获得DER编码的二进制信息：

更多资源请访问稀酷客(www.ckook.com)

```
// 返回此CRL的ASN.1 DER编码形式。
public abstract byte[] getEncoded()
// 从此CRL中获取以DER编码的CRL信息，即tbsCertList。
public abstract byte[] getTBSCertList()
```

以下是和时间信息有关的方法：

```
// 获取CRL的thisUpdate日期。
public abstract Date getThisUpdate()
// 获取CRL的nextUpdate日期。
public abstract Date getNextUpdate()
```

以下是和签名相关的方法：

```
// 获取CRL签名算法的签名算法名。
public abstract String getSigAlgName()
// 获取CRL的签名算法OID字符串。
public abstract String getSigAlgOID()
// 获取此CRL的签名算法中DER编码形式的签名算法参数。
public abstract byte[] getSigAlgParams()
```

通过上述方法，我们能够构建一个数字签名对象。

我们可以通过如下方法获得数字签名值：

```
// 获取CRL的signature值（原始签名位）。
public abstract byte[] getSignature()
```

可以通过以下方法获得X509CRLEntry的实例：

```
// 获取具有给定证书serialNumber的CRL项（如果有）。
public abstract X509CRLEntry getRevokedCertificate(BigInteger serialNumber)
// 获取此CRL中的所有项。
public abstract Set<? extends X509CRLEntry> getRevokedCertificates()
```

在Java 5中，X509CRL类实现了以下方法，用以获得X509CRLEntry的实例：

```
// 获取给定证书的CRL项（如果有）。
public X509CRLEntry getRevokedCertificate(X509Certificate certificate)
```

同时，我们可以通过以下方法校验CRL：

```
// 验证是否已使用与给定公钥相应的私钥签署了此CRL。
public abstract void verify(PublicKey key)
// 验证是否已使用与给定公钥相应的私钥签署了此CRL。
public abstract void verify(PublicKey key, String sigProvider)
```

此外，X509CRL类还提供了以下方法：

```
// 以X500Principal的形式返回CRL的发布方（发布方标识名）值。
public X500Principal getIssuerX500Principal()
```

❑ 实现示例

我和可以通过代码清单3-24获得证书撤销列表实体。

更多资源请访问稀酷客(www.ckook.com)

代码清单3-24 获得证书撤销列表实体

```
// 实例化，并指明证书类型为"X.509"。
CertificateFactory certificateFactory = CertificateFactory.getInstance("X.509");
// 获得证书输入流
FileInputStream in = new FileInputStream("D:\\x.keystore");
// 获得证书
X509Certificate certificate = (X509Certificate) certificateFactory.generateCertificate(in);
// 获得证书撤销列表
X509CRL x509CRL = (X509CRL) certificateFactory.generateCRL(in);
// 获得证书撤销列表实体
X509CRLEntry x509CRLEntry = x509CRL.getRevokedCertificate(certificate);
// 关闭流
in.close();
```

3.5.7 CertPath

CertPath类是一个抽象类，定义了常用于所有CertPath的方法。其子类可处理不同类型的证书（X.509、PGP等）。

所有CertPath对象都包含类型、Certificate列表及其支持的一种或多种编码。由于CertPath类是不可变的，所以构造CertPath后无法以任何外部可见的方式更改它。此规定适用于此类的所有公共字段和方法，以及由子类添加或重写的所有公共字段和方法。

```
// 不可变的证书序列（证书路径）。
public abstract class CertPath
extends Object
implements Serializable
```

❑ 方法详述

CertPath类要求子类实现如下方法。

我们可以通过以下方法获得该证书路径的证书列表：

```
// 返回此证书路径中的证书列表。
public abstract List<? extends Certificate> getCertificates()
```

以下方法可以获得证书路径二进制信息和编码信息：

```
// 返回此证书路径的编码形式，使用默认的编码。
public abstract byte[] getEncoded()
// 返回此证书路径的编码形式，使用指定的编码。
public abstract byte[] getEncoded(String encoding)
// 返回此证书路径支持的编码的迭代，默认编码方式优先。
public abstract Iterator<String> getEncodings()
```

CertPath类实现了以下方法获得证书类型：

```
// 返回此证书路径中的Certificate类型，如X.509。
public String getType()
```

此外，CertPath类覆盖了以下方法：

更多资源请访问稀酷客(www.ckook.com)

```
// 比较此证书路径与指定对象的相等性。
public boolean equals(Object other)
// 返回此证书路径的散列码。
public int hashCode()
// 返回此证书路径的字符串表示形式。
public String toString()
```

❑ 实现示例

我们可以通过代码清单3-25获得证书链。

代码清单3-25 获得证书链

```
// 实例化CertificateFactory对象，并指明证书类型为"X.509"。
CertificateFactory certificateFactory = CertificateFactory.getInstance("X.509");
// 获得证书输入流
FileInputStream in = new FileInputStream("D:\\x.keystore");
// 获得CertPath对象
CertPath certPath = certificateFactory.generateCertPath(in);
// 关闭流
in.close();
```

CertPath类作为证书链，它的操作离不开CertPathBuilder类和CertPathValidator类。有兴趣的读者请参考相应的Java API内容。

3.6 javax.net.ssl包详解

javax.net.ssl包提供用于安全套接字包的类。首先，作为本书加密与解密主题所涉及的最后一个话题，本节主要包含用于密钥和信任材料管理的工厂类（KeyManagerFactory类和TrustManagerFactory类）；其次，详述了用于表示安全套接字上下文的SSLContext类；最后，我们通过HttpsURLConnection类完成加密的网络通信。

3.6.1 KeyManagerFactory

KeyManagerFactory类是一个引擎类，它用来管理密钥，称为密钥管理工厂。

```
/* 此类充当基于密钥内容源的密钥管理器的工厂。每个密钥管理器管理特定类型的、由安全套接字所使用的密钥内容。密钥内容是基于KeyStore和/或提供者特定的源。*/
public class KeyManagerFactory
extends Object
```

❑ 方法详述

KeyManagerFactory类是引擎类，自然少不了通过getInstance()工厂方法完成对象实例化。我们可以通过指定算法名称获得实例化对象，方法如下所示：

```
// 返回充当密钥管理器工厂的KeyManagerFactory对象。
public final static KeyManagerFactory getInstance(String algorithm)
```

或者，指定算法名称的同时指定该算法的提供者：

更多资源请访问稀酷客(www.ckook.com)

```
// 返回充当密钥管理器工厂的KeyManagerFactory对象。
public final static KeyManagerFactory getInstance(String algorithm, Provider provider)
// 返回充当密钥管理器工厂的KeyManagerFactory对象。
public final static KeyManagerFactory getInstance(String algorithm, String provider)
```

当我们不知道该选用何种算法实例化对象时，可使用默认算法，方法如下所示：

```
// 获取默认的KeyManagerFactory算法名称。
public final static String getDefaultAlgorithm()
```

得到实例化对象后，可通过如下方法完成初始化：

```
// 使用密钥内容源初始化此KeyManagerFactory对象。
public final void init(KeyStore ks, char[] password)
// 使用特定于提供者的密钥内容源初始化此KeyManagerFactory对象。
public final void init(ManagerFactoryParameters spec)
```

我们可以通过如下方法获得密钥管理器数组：

```
// 为每类密钥内容返回一个密钥管理器。
public final KeyManager[] getKeyManagers()
```

KeyManagerFactory类是引擎类，同样提供了如下方法：

```
// 返回此KeyManagerFactory对象的提供者。
public final Provider getProvider()
// 返回此KeyManagerFactory对象的算法名称。
public final String getAlgorithm()
```

❑ 实现示例

我们可通过代码清单3-26构建密钥管理工厂。

代码清单3-26 构建密钥管理工厂

```
// 实例化KeyManagerFactory对象
KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance("SunX509");
// 加载密钥库文件
FileInputStream is = new FileInputStream("D:\\x.keystore");
// 实例化KeyStore对象
KeyStore ks = KeyStore.getInstance("JKS");
// 加载密钥库
ks.load(is, "password".toCharArray());
// 关闭流
is.close();
// 初始化KeyManagerFactory对象
keyManagerFactory.init(keyStore, "password".toCharArray());
```

获得密钥管理工厂后做什么用呢？我们将在后面介绍。

3.6.2 TrustManagerFactory

TrustManagerFactory类是用于管理信任材料的管理器工厂。

更多资源请访问稀酷客(www.ckook.com)

```
/* 此类充当基于信任材料源的信任管理器的工厂。每个信任管理器管理特定类型的由安全套接字使用的信任材料。信任材料是基于KeyStore和/或提供者特定的源。*/
public class TrustManagerFactory
extends Object
```

❑ 方法详述

TrustManagerFactory类的操作与KeyManagerFactory类很相似，同样需要getInstance()工厂方法获得实例化对象。

我们可以通过指定算法名称的方式获得实例化对象，方法如下所示：

```
// 返回充当信任管理器工厂的TrustManagerFactory对象。
public final static TrustManagerFactory getInstance(String algorithm)
```

或者，指定算法名称的同时指定该算法的提供者，方法如下所示：

```
// 返回充当信任管理器工厂的TrustManagerFactory对象。
public final static TrustManagerFactory getInstance(String algorithm, Provider provider)
// 返回充当信任管理器工厂的 TrustManagerFactory 对象。
public final static TrustManagerFactory getInstance(String algorithm, String provider)
```

当我们不知道该选用何种算法实例化对象时，可使用默认算法，方法如下所示：

```
// 获取默认的TrustManagerFactory算法名称。
public final static String getDefaultAlgorithm()
```

获得实例化对象后，需要通过以下方法完成初始化，这一点和KeyManagerFactory类如出一辙：

```
// 用证书授权源和相关的信任材料初始化此工厂。
public final void init(KeyStore ks)
// 使用特定于提供者的信任材料源初始化此工厂。
public final void init(ManagerFactoryParameters spec)
```

我们可以通过如下方法获得信任管理器数组：

```
// 为每种信任材料返回一个信任管理器。
public final TrustManager[] getTrustManagers()
```

TrustManagerFactory类是引擎类，同样提供了方法如下所示：

```
// 返回此TrustManagerFactory对象的算法名称。
public final String getAlgorithm()
// 返回此TrustManagerFactory对象的提供者。
public final Provider getProvider()
```

❑ 实现示例

我们可通过代码清单3-27构建信任管理工厂。

代码清单3-27 构建信任管理工厂

```
// 实例化TrustManagerFactory对象
TrustManagerFactory trustManagerFactory = TrustManagerFactory.getInstance("SunX509");
// 加载密钥库文件
```

更多资源请访问稀酷客(www.ckook.com)

```
FileInputStream is = new FileInputStream("D:\\x.keystore");
// 实例化KeyStore对象
KeyStore ks = KeyStore.getInstance("JKS");
// 加载密钥库
ks.load(is, "password".toCharArray());
// 关闭流
is.close();
// 初始化TrustManagerFactory对象
trustManagerFactory.init(trustkeyStore);
```

获得密钥管理工厂和信任材料管理工厂后做什么用呢？我们将在后面介绍。

3.6.3 SSLContext

SSLContext类用于表示安全套接字上下文，它同样是一个引擎类。

```
/* 此类的实例表示安全套接字协议的实现，它充当用于安全套接字工厂或 SSLEngine 的工厂。用可选的一组密钥和信任管理器及安全随机字节源初始化此类。*/
public class SSLContext
extends Object
```

❑ 方法详述

SSLContext类需要通过getInstance()工厂方法获得实例化对象。

一种最为常用的方式只需指定协议名称，方法如下所示：

```
// 返回实现指定安全套接字协议的SSLContext对象。
public static SSLContext getInstance(String protocol)
```

另一种方式需要指定协议名称的同时指定该协议的提供者，方法如下所示：

```
// 返回实现指定安全套接字协议的SSLContext对象。
public static SSLContext getInstance(String protocol, Provider provider)
// 返回实现指定安全套接字协议的SSLContext对象。
public static SSLContext getInstance(String protocol, String provider)
```

获得实例化对象后，需要通过以下方法完成初始化：

```
// 初始化此上下文。
publi c final void init(KeyManager[] km, TrustManager[] tm, SecureRandom random)
```

在完成初始化操作后，我们就可以获得该上下文中的属性，在本书中最为常用的是以下几种方法：

```
// 返回此上下文的ServerSocketFactory对象。
public final SSLServerSocketFactory getServerSocketFactory()
// 返回此上下文的SocketFactory对象。

public final SSLSocketFactory getSocketFactory()
```

有关SSLSessionContext类相关内容，读者可以参考相关Java API文档，以下方法提供了获得服务器端/客户端SSLSessionContext对象实现：

```
/* 返回服务器会话上下文，它表示可供服务器端SSL套接字握手阶段所使用的SSL会话集。*/
```

更多资源请访问稀酷客(www.ckook.com)

```
public final SSLSessionContext getServerSessionContext()
/* 返回客户端会话上下文，它表示可供客户端SSL套接字握手阶段所使用的SSL会话集。*/
public final SSLSessionContext getClientSessionContext()
```

有关SSLEngine类的相关内容，读者可以参考相关的Java API文档，以下方法提供了创建SSLEngine对象实现：

```
// 使用此上下文创建新的SSLEngine。
public final SSLEngine createSSLEngine()
// 使用此上下文创建新的SSLEngine，并绑定主机和端口。
public final SSLEngine createSSLEngine(String peerHost, int peerPort)
```

以下方法用于设置/获得默认SSL上下文：

```
// 设置默认的SSL上下文
public synchronized static void setDefault(SSLContext context)
// 返回默认的SSL上下文
public synchronized static SSLContext getDefault()
```

以下方法用于设置/获得默认SSL参数：

```
// 返回表示此SSL上下文默认设置的SSLParameters的副本。
public final SSLParameters getDefaultSSLParameters()
// 返回表示此SSL上下文受支持设置的SSLParameters的副本。
public final SSLParameters getSupportedSSLParameters()
```

SSLContext类作为引擎类，提供了以下方法：

```
// 返回此SSLContext对象的协议名称。
public final String getProtocol()
// 返回此SSLContext对象的提供者。
public final Provider getProvider()
```

❑ 实现示例

我们通过代码清单3-28来展示如何使用KeyMangagerFactory、TrustmanagerFactory、SSLContext和SSLSocektFactory类。

代码清单3-28 构建SSLSocketFactory

```
/**
* 获得KeyStore
*
* @param keyStorePath
* @param password
* @return
* @throws Exception
*/
private static KeyStore getKeyStore(String keyStorePath, String password) throws Exception {
    // 获得密钥库文件输入流
    FileInputStream is = new FileInputStream(keyStorePath);
    // 实例化密钥库
    KeyStore ks = KeyStore.getInstance("JKS");
```

更多资源请访问稀酷客(www.ckook.com)

```
        // 加载密钥库
        ks.load(is, password.toCharArray());
        // 关闭流
        is.close();
        return ks;
    }
    /**
    * 获得SSLSocektFactory
    * @param password
    *              密码
    * @param keyStorePath
    *              密钥库路径
    * @param trustKeyStorePath
    *              信任库路径
    * @return
    * @throws Exception
    */
    private static SSLSocketFactory getSSLSocketFactory(String password, String
    keyStorePath, String trustKeyStorePath) throws Exception {
        // 初始化密钥库
        KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance
        ("SunX509");
        KeyStore keyStore = getKeyStore(keyStorePath, password);
        keyManagerFactory.init(keyStore, password.toCharArray());
        // 初始化信任库
        TrustManagerFactory trustManagerFactory = TrustManagerFactory.getInstance
        ("SunX509");
        KeyStore trustkeyStore = getKeyStore(trustKeyStorePath, password);
        trustManagerFactory.init(trustkeyStore);
        // 初始化SSL上下文
        SSLContext ctx = SSLContext.getInstance("SSL");
        ctx.init(keyManagerFactory.getKeyManagers(),trustManagerFactory.
        getTrustManagers(), null);
        SSLSocketFactory sf = ctx.getSocketFactory();
        return sf;
    }
```

至此，我们完成了构建HTTPS协议的准备工作。接下来我们将构建一个基于HTTPS协议的网络连接。

3.6.4 HttpsURLConnection

HttpsURLConnection类继承于HttpURLConnection类。从字面上看，两个类仅差一个字母。但在含义上，HttpsURLConnection类比HttpURLConnection类更具安全性。

```
/* HttpsURLConnection扩展HttpURLConnection，支持各种特定于https功能。*/
public abstract class HttpsURLConnection
extends HttpURLConnection
```

更多资源请访问稀酷客(www.ckook.com)

❑ 方法详述

HttpsURLConnection类的方法有很多，但对于本书将要阐述的内容来讲，主要用到了如下几种方法。

我们可以通过如下方法设置（默认）SSLSocketFactory对象：

```
/* 设置当此实例为安全https URL连接创建套接字时使用的SSLSocketFactory。*/
public void setSSLSocketFactory(SSLSocketFactory sf)
// 设置此类的新实例所继承的默认SSLSocketFactory。
public static void setDefaultSSLSocketFactory(SSLSocketFactory sf)
```

相应地，我们可以通过如下方法获得（默认）SSLSocketFactory对象：

```
// 获取为安全https URL连接创建套接字时使用的SSL套接字工厂。
public SSLSocketFactory getSSLSocketFactory()
// 获取此类的新实例所继承的默认静态SSLSocketFactory。
public static SSLSocketFactory getDefaultSSLSocketFactory()
```

我们可以通过如下方法获得握手期间相关的证书链：

```
// 返回握手期间发送给服务器的证书。
public abstract Certificate[] getLocalCertificates()
// 返回服务器的证书链，它是作为定义会话的一部分而建立的。
public abstract Certificate[] getServerCertificates()
```

关于Principal类，请读者参考相应的Java API文档，以下是HttpsURLConnection类提供的相应方法：

```
// 返回握手期间发送到服务器的主体。
public Principal getLocalPrincipal()
// 返回服务器的主体，它是作为定义会话的一部分而建立的。
public Principal getPeerPrincipal()
```

以下方法用于获得密码套件：

```
// 返回在此连接上使用的密码套件。
public abstract String getCipherSuite()
```

关于HostnameVerifier类，请读者朋友参考相应的Java API文档，以下方法用于获得（默认）HostnameVerifier对象：

```
// 获取此类的新实例所继承的默认HostnameVerifier。
public static HostnameVerifier getDefaultHostnameVerifier()
// 获取此实例适当的HostnameVerifier。
public HostnameVerifier getHostnameVerifier()
```

以下方法用于获得（默认）HostnameVerifier对象：

```
// 设置此类的新实例所继承的默认HostnameVerifier。
public static void setDefaultHostnameVerifier(HostnameVerifier v)
// 设置此实例的HostnameVerifier。
```

更多资源请访问稀酷客(www.ckook.com)

```
public void setHostnameVerifier(HostnameVerifier v)
```

❑ 实现示例

我们接3.6.3节内容，获得SSLSocketFactory对象后，可完成HttpsURLConnection对象的设置，如代码清单3-29所示：

代码清单3-29 构建HTTPS

```
// 构建URL对象
URL url = new URL("https://www.sun.com/");
// 获得HttpsURLConnection实例化对象。
HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
// 打开输入模式
conn.setDoInput(true);
// 打开输出模式
conn.setDoOutput(true);
// 在这里调用前面介绍的configSSLSocketFactory()方法。
// 设置SSLSocketFactory
configSSLSocketFactory(conn, "password", "D:\\x.keystore", " D:\\x.keystore ");
// 获得输入流
InputStream is = conn.getInputStream();
// 若正常打开Https，将获得一个有效值（即contentLength的值不为-1）。
int length = conn.getContentLength();
// ……
// 关闭流
is.close();
```

至此，我们就可以通过带有证书的HTTPS连接进行消息传递了。

3.7 小结

在本章内容的学习中，我们了解了有关Java安全领域的内容。它共分为三个部分：JCE（Java Cryptography Extension，Java加密扩展包）、JSSE（Java Secure Sockets Extension，Java安全套接字扩展包）、JAAS（Java Authentication and Authentication Service，Java鉴别与安全服务）。JCE和JSSE是本书中主要阐述的重点内容。

在安全提供者体系结构中，我们认识了Provider类和Security类，它们共同构成了安全提供者的概念。在Java 6版本中，共配置了9种安全提供者。它们均是Provider类的子类。Security类则用于管理这些提供者。

受军事出口的限制，密码算法的实现强度有所限制。如DES算法受出口限制，其密钥长度为56位，而实际上出于安全考虑则要求密钥长度为128位。这种安全强度已不能满足当前应用环境的需要。

在java.security包和javax.crypto包中，我们详述了如何构建对称密钥和非对称密钥，以及如何使用加密组件实现加密算法。

更多资源请访问稀酷客(www.ckook.com)

如Key接口，作为顶层密钥接口定义了密钥的基本操作。SecretKey、PublicKey和PrivateKey接口均是Key接口的子接口，分别定义了用于对称加密算法的秘密密钥和用于非对称加密算法的公钥/私钥。

我们可以通过相应的生成器和工厂类完成密钥的生成，如KeyGenerator和SecretKeyFactory用于秘密密钥的生成；KeyPairGenertor和KeyFactory用于公钥/私钥的生成。

通过Cipher及其扩展类CipherInputStream和CipherOutputStream，可以完成加密与解密算法的实现。通过Sinature类则可以完成签名与验证操作。

除此之外，我们还了解了消息摘要算法的两种实现方式：Mac类，用于实现安全消息摘要算法；MessageDigest类及其扩展类DigestInputStream和DigestOutputStream，用于实现一般消息摘要算法。

对于密钥管理，我们提到了KeyStore类。它将在密钥管理和证书管理的操作中，多次使用到。

在java.security.spec包和javax.crypto.spec包中主要详述如何通过密钥规范还原密钥对象。我们将会多次使用到这两个包中的实现。密钥通常以二进制的方式存储在文件中，通过这两个包中的实现我们将可以将二进制数据还原为秘密密钥、公钥和私钥对象。

在java.security.cert包中，我们主要详述了如何通过CertificateFactory类构建证书（Certificate）对象和证书撤销列表（CRL）对象。在获得证书对象后，我们可以获得相应的算法、公钥等信息。这将有助于我们完成相应的加密解密操作以及数字签名与验证操作。

在了解了上述与加密与解密算法相关的实现，以及与数字证书操作有关的实现内容后，我们来关注javax.net.ssl包中的内容。在这一节中，我们最终达到了通过证书构建基于HTTPS协议的加密网络通信。

Java是一门不断发展的语言，所涉及的领域越来越多。网络应用、电子商务等领域都促进着Java加密与解密技术的发展，我们将在今后的开发过程中更加频繁地使用到Java所提供的安全实现。

更多资源请访问稀酷客(www.ckook.com)

第4章

他山之石，可以攻玉

在第3章中，我们了解了Java 6中与加密/解密相关的API，几乎各种常用加密算法都能找到对应的实现，但还是难免会有遗憾：受出口限制，密钥长度上不能满足需求；部分算法未能支持，如MD4、SHA-224等算法；API使用起来还不是很方便；一些常用的进制转换辅助工具未能提供，如Base64编码转换、十六进制编码转换等工具。

对于上述这些问题，该如何解决呢？他山之石，可以攻玉！

对于出口限制，Sun通过权限文件（local_policy.jar、US_export_policy.jar）做了相应限制。但幸运的是，Sun在官方主页上提供了替换文件，可减少相关的限制。当然，你需要结合本地进出口政策，合法使用该文件。

对于Java 6未能支持的加密算法，各大密码学机构以及以加密算法为核心的软件组织和软件公司都不遗余力地研究，并提供了相应的实现。但因为版权问题，我们很少能够使用到这些机构提供的加密软件包。在这样一个开源的时代，难道就没有其他的替代方案吗？当然不是，Bouncy Castle（http://www.bouncycastle.org/）提供了一系列算法支持实现，并可以跻身于JCE框架之下，以提供者的方式纳入其中。而且，Bouncy Castle是一款开源组件，你不必为版权问题而伤脑筋。此外，Bouncy Castle提供了Base64和十六进制编码转换相关的实现。

Commons Codec（http://commons.apache.org/codec/）是国际开源组织Apache（http://www.apache.org/）旗下的一款开源软件。它与Bouncy Castle不同，并未对Java 6提供扩展加密算法。仅仅是对Java 6提供的API做了改进，提供了更加易用的API。

4.1 加固你的系统

鉴于出口限制问题，我们得到的JDK安全强度不够高，主要是密钥长度不够。对于这一点，Sun在其官方网站上提供了无政策限制权限文件（Unlimited Strength Jurisdiction Policy Files），我们只需要将其部署在JRE环境中。

如果你所在的国家或地区不受该进出口限制，就完全可以使用该文件解除权限限制问题。

更多资源请访问稀酷客(www.ckook.com)

4.1.1 获得权限文件

Sun在其下载页面（http://java.sun.com/javase/downloads/index.jsp）上提供了该权限文件的下载地址，如图4-1所示，注意Other Downloads项。

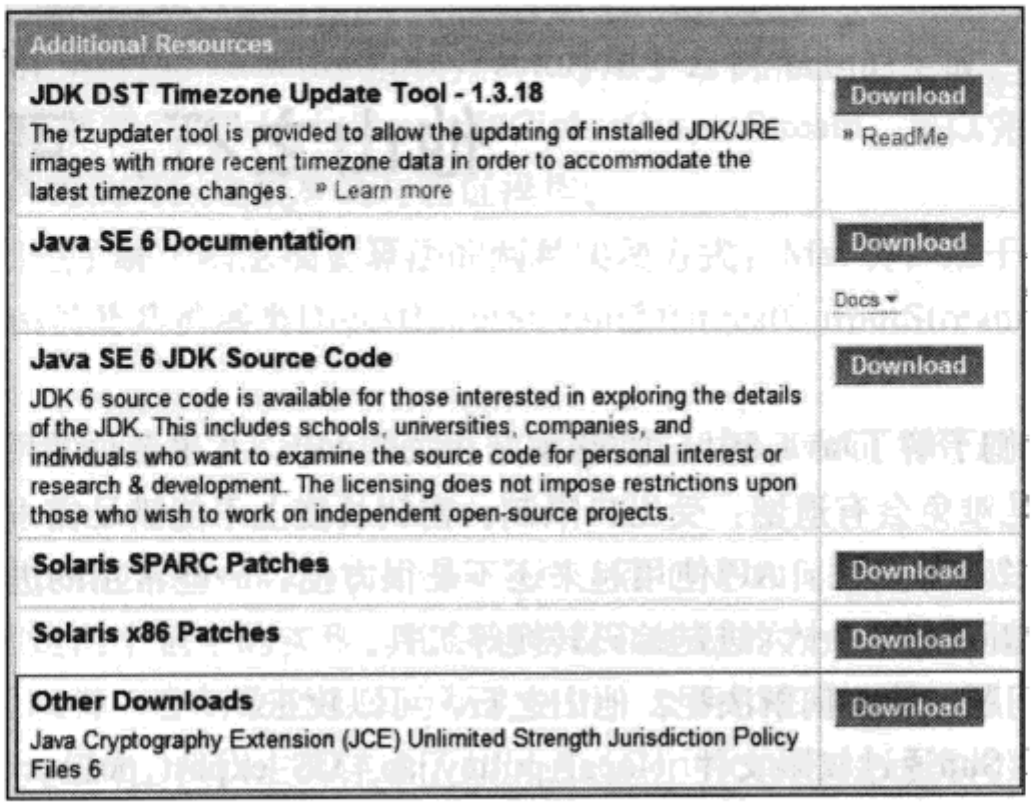


图4-1 权限文件下载页面

Other Downloads项提供了Java Cryptography Extension（JCE）Unlimited Strength Jurisdiction Policy Files 6的下载地址。很显然，它的版本是6，对应Java 6。下载后，会得到一个名为jce_policy-6.zip的文件。用WinRAR打开后，如图4-2所示。

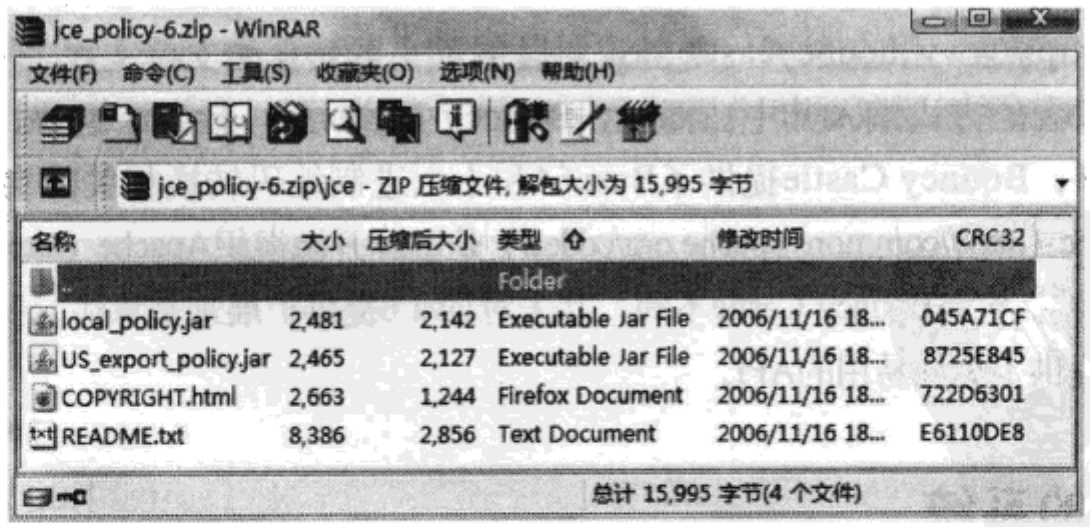


图4-2 jce_policy-6.zip文件中jce目录列表

在这个压缩包中仅有一个目录，也就是jce目录。该目录中包含了4个文件：README.txt、COPYRIGHT.html、local_policy.jar和US_export_policy.jar。其中包含的两个jar文件正是此次配置中用到的文件。

4.1.2 配置权限文件

我们可以查看上述README.txt文件，你需要在JDK的JRE环境中，或者是JRE环境中配置

更多资源请访问稀酷客(www.ckook.com)

上述两个jar文件。

切换到%JDK_Home%\jre\lib\security目录下，对应覆盖local_policy.jar和US_export_policy.jar两个文件。同时，你可能有必要在%JRE_Home%\lib\security目录下，也需要对应覆盖这两个文件。

配置权限文件的最终目的是为了使应用在运行环境中获得相应的权限，可以加强应用的安全性。

通常，我们在应用服务器上安装的是JRE，而不是JDK。因此，这就很有必要在应用服务器的%JRE_Home%\lib\security目录下，对应覆盖这两个权限文件。很多开发人员往往忽略了这一点，导致事故发生。

4.1.3 验证配置

经过一番调整之后，如何验证我们的系统获得相应的权限呢？

修改权限配置的目的是为了获得更长的密钥。如果对于同一个加密算法，在修改权限配置前后系统所能提供的密钥的最长长度发生了变化，那么就说明配置有效！

AES算法是较为常用的对称加密算法之一，几乎是对称加密算法中安全级别最高的算法。Java 6支持AES算法的密钥长度为128位、192位或256位。但是，如果不加权限配置直接使用256位长度的密钥，就会得到“java.security.InvalidKeyException”异常，如下面代码所示：

```
KeyGenerator kg = KeyGenerator.getInstance(KEY_ALGORITHM);
// AES 要求密钥长度为128位、192位或256位。
kg.init(256);
// 生成秘密密钥
SecretKey secretKey = kg.generateKey();
byte[] key = secretKey.getEncoded();
```

如果我们未能对相应的权限配置文件作相应调整，则会得到“java.security.InvalidKeyException”异常。反之，将正常获得256位长度的密钥！

有关AES算法相关实现细节，请读者阅读第7章。

4.2 加密组件Bouncy Castle

Java 6提供了多种算法支持，但并不完善。许多加密强度较高的算法，Java 6未能提供。在本书第3章中，我们提到了java.security文件，它位于%JDK_HOME%\jre\lib\security\ext目录下，用于提供者配置。

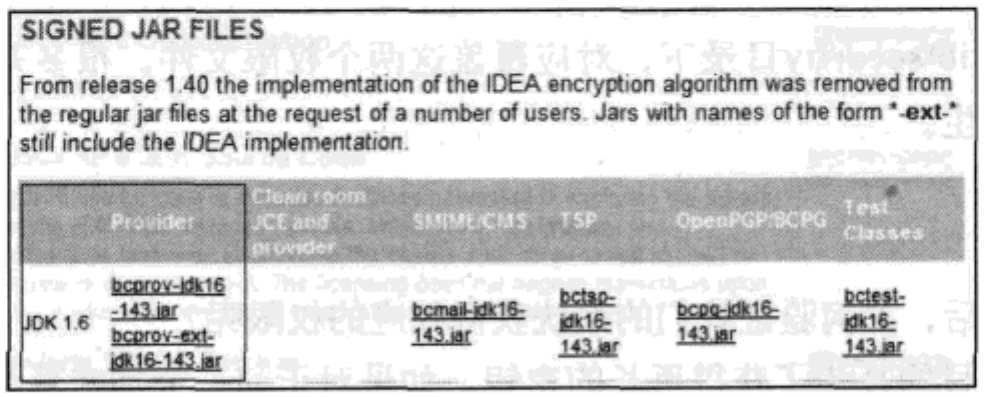
如果你需要使用Java 6不支持的算法，如MD4和IDEA（International Data Encryption Algorithm，国际数据加密算法）等。你可以在继续沿用Java 6的API前提下，通过在JRE环境中配置开源组件包Bouncy Castle，加入对应的提供者，获得相应的算法支持。关于Bouncy Castle支持的算法，请参见附录。

更多资源请访问稀酷客(www.ckook.com)

4.2.1 获得加密组件

Bouncy Castle目前提供的加密组件包的版本是1.43。自1.40版本开始，Bouncy Castle提供了对IDEA算法的支持。

我们可以通过Bouncy Castle提供的下载地址（http://www.bouncycastle.org/latest_releases.html），下载最新的加密组件包，主要是bcprov-jdk16-143.jar和bcprov-ext-jdk16-143.jar两个文件，如图4-3所示。

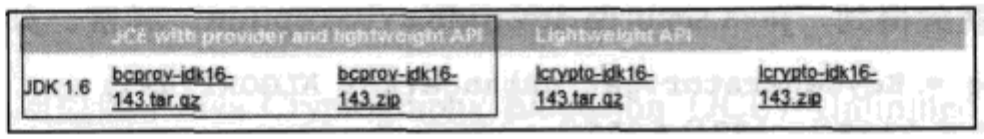

SIGNED JAR FILES

From release 1.40 the implementation of the IDEA encryption algorithm was removed from the regular jar files at the request of a number of users. Jars with names of the form *-ext-* still include the IDEA implementation.

	Provider	Clean room JCE and provider	S/MIME/CMS	TSP	OpenPGP/BCPG	Test Classes
JDK 1.6	bcprov-jdk16-143.jar bcprov-ext-jdk16-143.jar		bcmail-jdk16-143.jar	bctsp-jdk16-143.jar	bcpg-jdk16-143.jar	bctest-jdk16-143.jar

图4-3 Bouncy Castle可执行二进制文件下载页面

当然，你有可能需要相应的源代码和API文档，它位于bcprov-jdk16-143.zip文件中，如图4-4所示。



	JCE with provider and lightweight API		Lightweight API	
JDK 1.6	bcprov-jdk16-143.tar.gz	bcprov-jdk16-143.zip	lcrypto-jdk16-143.tar.gz	lcrypto-jdk16-143.zip

图4-4 Bouncy Castle文档及源代码文件下载页面

获得bcprov-jdk16-143.jar和bcprov-ext-jdk16-143.jar两个文件后，我们就可以在Java 6的基础上扩充算法支持了。后面详细讲解这两个文件的使用方式。

Bouncy Castle加密组件包支持的算法不仅广泛，而且开源，常常用做J2ME的加密实现，其主页上也提供了J2ME版本的下载链接（http://www.bouncycastle.org/download/lcrypto-j2me-143.zip）。

4.2.2 扩充算法支持

对于Bouncy Castle提供的扩充算法支持，我们有两种方案可选：

1）配置方式。通过配置JRE环境，使其作为提供者（Provider）提供相应的算法支持，在代码实现层面只需指定要扩展的算法名称。

2）调用方式。在调用Java API初始化相应的密钥工厂、密钥生成器等引擎类之前，通过代码将Bouncy Castle提供者引入，获得扩展算法支持。

1. 配置方式

配置Bouncy Castle并不复杂，有点类似于4.1节中权限文件的配置，需要在%JDK_Home%和%JRE_Home%目录中做相应调整。

我们以%JDK_Home%目录配置为例。

更多资源请访问稀酷客(www.ckook.com)

❑ 使用步骤

首先，我们需要修改配置文件（java.security）。

在第3章中提到配置%JDK_Home%\ jre\lib\security\java.security文件，通过加入支持的方式获得更多的算法支持。

在这个文件中，我们可以很清晰地看到Java 6中有如下9种安全提供者：

```
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=com.sun.net.ssl.internal.ssl.Provider
security.provider.4=com.sun.crypto.provider.SunJCE
security.provider.5=sun.security.jgss.SunProvider
security.provider.6=com.sun.security.sasl.Provider
security.provider.7=org.jcp.xml.dsig.internal.dom.XMLDSigRI
security.provider.8=sun.security.smartcardio.SunPCSC
security.provider.9=sun.security.mscapi.SunMSCAPI
```

Java 7很快就要问世了，作者查看了该文件的相应配置，并无变化。

上述这些配置是按照以下方式来配置的：

```
#security.provider.<n>=<className>
```

很显然，为了加入Bouncy Castle加密组件的安全提供者只需要这样做：

```
#增加BouncyCastleProvider
security.provider.10=org.bouncycastle.jce.provider.BouncyCastleProvider
```

最后，我们需要将bcprov-ext-jdk16-143.jar文件导入。

切换至%JDK_Home%\jre\lib\ext目录下，我们能够看到 sunjce_provider.jar这个文件。SunJCE就是由这个文件提供的。同理，要将Bouncy Castle加密组件扩展包导入其中，只需要将4.2.1节中获得的bcprov-ext-jdk16-143.jar文件放到这里即可。

%JRE_Home%目录的相应配置与上述%JDK_Home%目录配置相类似。

对应修改%JRE_Home%\lib\security\java.security文件，并将bcprov-ext-jdk16-143.jar文件放置到%JRE_Home%\lib\ext目录中即可。

❑ 应用举例

Java 6不支持MD4算法，做了上述配置后，如果要使用MD4算法可参考如下代码：

```
/**
* MD4加密
* @param data
* @return
* @throws Exception
*/
public static byte[] encodeMD4(byte[] data) throws Exception {
    MessageDigest md = MessageDigest.getInstance("MD4");
    md.update(data);
    return md.digest();
}
```

更多资源请访问稀酷客(www.ckook.com)

这是一种对使用者透明的使用方式，你无须关心MD4算法的提供者是谁，代码很清晰。

2. 调用方式

有时候，我们需要通过明显的代码调用方式引入支持者，这完全依赖于Security类的addProvider()方法，详见3.2节。

❑ 使用步骤

首先，我们需要将bcprov-jdk16-143.jar文件导入工程。相信读者对于这一步操作一定都不陌生，这里就不详细介绍了。

接着，我们需要将以下两个类导入（import）你的代码中：

```
import java.security.Security;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
```

当然，如果你使用Eclipse，可以在下述代码写完后，使用快捷键Ctrl+Shift+O直接导入所需类。

最后，我们只需要在初始化密钥工厂、密钥生成器等引擎类之前，调用如下代码：

```
// 加入BouncyCastleProvider支持
Security.addProvider(new BouncyCastleProvider());
```

或者，在初始化密钥工厂、密钥生成器等引擎类时，采用如下方式：

```
MessageDigest md = MessageDigest.getInstance("MD4", "BC");
```

每个提供者都有简称，Bouncy Castle提供者简称“BC”，因此我们可以通过上述方式使用BouncyCastleProvider。

执行以下代码，我们可以获得Bouncy Castle提供者的算法详细描述。

```
Provider provider = Security.getProvider("BC");
System.err.println(provider);
for (Map.Entry<Object, Object> entry : provider.entrySet()) {
    System.err.println(entry.getKey() + " - " + entry.getValue());
}
```

在控制台中我们可以看到以下内容：

```
BC version 1.43
    Alg.Alias.Signature.SHA224withCVC-ECDSA - SHA224WITHCVC-ECDSA
    AlgorithmParameters.DES - org.bouncycastle.jce.provider.JDKAlgorithm
    Parameters$IVAlgorithmParameters
    KeyGenerator.2.16.840.1.101.3.4.22 - org.bouncycastle.jce.provider.
    symmetric.AES$KeyGen192
    Alg.Alias.Cipher.RSA//ISO9796-1PADDING - RSA/ISO9796-1
    AlgorithmParameterGenerator.NOEKEON - org.bouncycastle.jce.provider.
    symmetric.Noekeon$AlgParamGen
    Alg.Alias.Cipher.RSA//NOPADDING - RSA
```

更多资源请访问稀酷客(www.ckook.com)

```
    Alg.Alias.Mac.IDEA - IDEAMAC
    Alg.Alias.AlgorithmParameters.PBEWITHSHAAND3-KEYTRIPLEDES - PKCS12PBE
    Alg.Alias.Mac.HMAC/SHA1 - HMACSHA1
    Alg.Alias.AlgorithmParameterGenerator.1.2.410.200004.1.4 - SEED
    AlgorithmParameterGenerator.ELGAMAL - org.bouncycastle.jce.provider.
    JDKAlgorithmParameterGenerator$ElGamal
.....
```

上述内容并未完全展现Bouncy Castle所支持的算法，本文做了简要摘录。

通常我们也可以通过上述方式检查系统支持的加密算法。

❑ 应用举例

Java 6未能支持MD4算法，也未能支持SHA-224算法。依照本文显式调用代码的方式，需要将bcprov-jdk16-143.jar文件导入工程，同时导入相关类（Security和BouncyCastleProvider），并通过Security类的addProvider()方法将BouncyCastleProvider类导入，见如下代码：

```
import java.security.Security;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
// ... 省略
/**
* SHA-224加密
*
* @param data
* @return
* @throws Exception
*/
public static byte[] encodeSHA224(byte[] data) throws Exception {
    // 加入BouncyCastleProvider支持
    Security.addProvider(new BouncyCastleProvider());
    MessageDigest md = MessageDigest.getInstance("SHA-224");
    md.update(data);
    return md.digest();
}
```

多了几行代码，多少有点别扭。但这种方式让人心知肚明，很清楚自己使用了哪些类。

3. 两种方式对比

配置和调用两种方法都有可取之处：前者对代码无需改动，只需提供Bouncy Castle支持的算法名称，但开发者需要知道Bouncy Castle已作为提供者部署在JRE中，也就是说这种方式需要依赖环境；后者需要对代码做改动，将Bouncy Castle作为提供者在代码中调用，但对环境依赖程度较小。

作者对于这两种方式并没有一个绝对的评价，它们各有特色。读者可根据需要，选择合适的方式获得扩展算法支持。

为了引起读者的注意，避免不必要的误导，本文在后续内容中将采用第2种方式介绍Bouncy Castle相关扩展算法。

更多资源请访问稀酷客(www.ckook.com)

4.2.3 相关API

Bouncy Castle的API主要包含了以下几个方面：

❑ JCE工具及其扩展包

仅包括org.bouncycastle.jce包。这是对JCE框架的支持。其中定义了一些扩展算法的接口与实现，如ECC和ElGamal算法。

❑ JCE支持者和测试包

包括org.bouncycastle.jce.provider包及其子包。本章提到的Bouncy Castle 的安全提供者BouncyCastleProvider就位于该包中。

❑ 轻量级加密包

包括org.bouncycastle.crypto包及其子包。我们可以认为这个包完成了扩展算法的实现。

❑ OCSP和OpenSSL PEM支持包

包括org.bouncycastle.ocsp包及其子包和org.bouncycastle.openssl包及其子包。这两个包都是与数字证书相关的支持包。OCSP（Online Certificate Status Protocol，在线证书状态协议）用于鉴定所需证书的（撤销）状态。具体协议内容请查看RFC 2560（http://www.ietf.org/rfc/rfc2560.txt）。OpenSSL用于管理数字证书，包括证书的申请和撤销等。

❑ ASN.1编码支持包

包括org.bouncycastle.asn1包及其子包，该包体积最为庞大。标准的ASN.1编码规则有基本编码规则（Basic Encoding Rules, BER）、规范编码规则（Canonical Encoding Rules, CER）、唯一编码规则（Distinguished Encoding Rules, DER）、压缩编码规则（Packed Encoding Rules, PER）和XML编码规则（XML Encoding Rules, XER）。我们在导出数字证书时，常常会使用到DER编码。

❑ 工具包

包括org.bouncycastle.util包及其子包。提供了很多与编码转换有关的工具类，如Base64编码和十六进制编码。

❑ 其他包

包括org.bouncycastle.mozilla包及其子包和org.bouncycastle.x509包及其子包。org.bouncycastle.mozilla包用于支持基于Mozilla（网景）浏览器的公钥签名和身份认证。org.bouncycastle.x509包用于基于支持X.509格式的数字证书。

相信不论是谁，在看了上述介绍后都会汗颜。当然，作者也不例外。作者有意避开Bouncy Castle支系庞大的API，建议读者将Bouncy Castle作为支持的方式来使用其扩展算法。这样可以降低学习难度，这也是Sun构建JCE架构最初的目的。

本书将简要介绍org.bouncycastle.util.encoders包中的内容。

1. Base64

Base64类是用于Base64编码的工具类。当然，Sun内部实现了Base64算法，但相关实现并未在API中体现，请读者参考第5章的内容。

更多资源请访问稀酷客(www.ckook.com)

```
// 用于Base64编码/解码转换。
public class Base64
extends Object
```

❑ 方法详述

我们先来看编码方法：

```
// 返回Base64编码结果。
public static byte[] encode(byte[] data)
// 向输出流中写入Base64编码结果。
public static int encode(byte[] data, int off, int length, OutputStream out)
// 向输出流中写入Base64编码结果。
public static int encode(byte[] data, OutputStream out)
```

以下是对应的解码方法：

```
// 返回Base64解码结果。
public static byte[] decode(byte[] data)
// 返回Base64解码结果，空格将被忽略。
public static byte[] decode(String data)
// 向输出流中写入Base64解码结果，空格将被忽略。
public static int decode(String data, OutputStream out)
```

其实，Base64类在内部实现时调用了Base64Encoder类来完成相应的编码/解码操作。我们只需要知道如何使用Base64类就可以了。

❑ 实现示例

我们通过代码清单4-1来演示如何进行Base64编码和解码操作。

代码清单4-1 Base64编码/解码1

```
String str = "base64编码";
System.err.println("原文:\t" + str);
byte[] input = str.getBytes();
// Base64编码
byte[] data = Base64.encode(input);
System.err.println("编码后:\t" + new String(data));
// Base64解码
byte[] output = Base64.decode(data);
System.err.println("解码后:\t" + new String(output));
```

查看控制台输出如下：

```
原文:       Base64 编码
编码后:     QmFzZTY0IOe8lueggQ==
解码后:     Base64 编码
```

编码后的内容中，出现“=”符号，这是Base64编码的标志性符号。

2. UrlBase64

Base64算法最初用于电子邮件系统，后经演变成为显式传递Url参数的一种编码算法，通

更多资源请访问稀酷客(www.ckook.com)

常称为“Url Base64”。它是Base64算法的变体，将字符映射表中用做补位的“=”换成“.”，并用“-”和“_”分别替代“+”和“/”，使得Base64编码符合Url参数规则，可以将二进制的数据以Get方式进行传输。有关Base64算法请参见第5章。

```
// 用于Url Base64编码/解码转换。
public class UrlBase64
extends Object
```

❑ 方法详述

以下为编码方法：

```
// 返回Url Base64编码。
public static byte[] encode(byte[] data)
```

以下方法将编码结果输出至输出流中：

```
// 向输出流中写入Url Base64编码。
public static int encode(byte[] data, OutputStream out)
```

以下为解码方法：

```
// 返回Url Base64解码，空格忽略。
public static byte[] decode(byte[] data)
// 返回Url Base64解码，空格忽略。
public static byte[] decode(String data)
```

以下方法可以将解码结果输出至输出流中：

```
// 向输出流中写入Url Base64解码，空格忽略。
public static int decode(byte[] data, OutputStream out)
// 向输出流中写入Url Base64解码，空格忽略。
public static int decode(String data, OutputStream out)
```

UrlBase64类在内部实现时，调用了UrlBase64Encoder类来完成相应的编码/解码操作。这是Bouncy Castle一贯的代码风格。

❑ 实现示例

我们对上述Base64算法实现稍作调整，改为Url Base64类完成编码和解码操作，如代码清单4-2所示。

代码清单4-2 Url Base64编码/解码

```
String str = "Base64 编码";
System.err.println("原文:\t" + str);
byte[] input = str.getBytes();
// Url Base64编码
byte[] data = UrlBase64.encode(input);
System.err.println("编码后:\t" + new String(data));
// Url Base64解码
byte[] output = UrlBase64.decode(data);
System.err.println("解码后:\t" + new String(output));
```

更多资源请访问稀酷客(www.ckook.com)

观察控制台输出的内容：

```
原文：      Base64编码
编码后：    QmFzZTY0IOe8lueggQ..
解码后：    Url Base64编码
```

编码后的内容中，出现了“.”符号，替换掉了“=”符号，符合了Url参数规则。当然，有些系统会对这个“.”符号较为敏感，如在文件系统中这可能导致一些错误。关于这个问题，请读者参考第5章的相关内容。

3. Hex

不用介绍，读者就能从字面上判断出Hex类和十六进制相关。Hex类用于十六进制转换。常配合消息摘要算法处理摘要值，以十六进制形式公示。

```
// 用于十六进制编码/解码操作。
public class Hex
extends Object
```

❑ 方法详述

以下是十六进制编码方法：

```
// 返回十六进制编码结果。
public static byte[] encode(byte[] data)
// 返回十六进制编码结果。
public static byte[] encode(byte[] data, int off, int length)
```

以下方法将编码结果输出至输出流中：

```
// 向输出流中写入十六进制编码结果。
public static int encode(byte[] data, int off, int length, OutputStream out)
// 向输出流中写入十六进制编码结果。
public static int encode(byte[] data, OutputStream out)
```

以下是十六进制解码方法：

```
// 返回十六进制解码结果。
public static byte[] decode(byte[] data)
// 返回十六进制解码结果，空格忽略。
public static byte[] decode(String data)
```

以下方法将解码结果输出至输出流中：

```
// 向输出流中写入十六进制解码结果，空格忽略。
public static int decode(String data, OutputStream out)
```

❑ 实现示例

我们继续对上述代码进行改造，如代码清单4-3所示：

代码清单4-3 Hex编码/解码1

```
String str = "Hex 编码";
System.err.println("原文:\t" + str);
```

更多资源请访问稀酷客(www.ckook.com)

```
byte[] input = str.getBytes();
// Hex编码。
byte[] data = Hex.encode(input);
System.err.println("编码后:\t" + new String(data));
// Hex解码。
byte[] output = Hex.decode(data);
System.err.println("解码后:\t" + new String(output));
```

观察控制台输出的内容：

```
原文:       Hex编码
编码后:     48657820e7bc96e7a081
解码后:     Hex编码
```

这时候看到编码后的字符串样式就很眼熟了，在各大软件厂商的下载页面上都很常见，通常用做MD5的十六进制表示。

Sun虽然在Java API中提供了简单的进制转换方法，但并不方便。相应的进制转换方法均包含在封装类中，如Long类的toHexString()方法可将长整型转换为十六进制字符串，并通过parseLong()方法，指定进制参数将十六进制字符串转换为长整型，参考代码如下：

```
// 长整型转换十六进制字符串。
String s = Long.toHexString(new Long(100));
// 十六进制字符串转换长整型。
long l = Long.parseLong(s, 16);
```

如果输入参数为字节数组，就不能使用上述方式来操作了。

对于一个长整型、整型、浮点型等数据类型来说，二进制转换虽然不复杂，我们完全有能力实现，但是转换过程中要考虑的细节很多，往往容易在补码、转码等问题上出现纰漏，这时就不如使用现成的开源实现。也许“不要重复制造轮子”的经典论断就是这么来的。

4.3 辅助工具Commons Codec

Commons Codec是Apache旗下的一款开源软件，主要用于编码格式的转换，如Base64、二进制、十六进制、字符集和Url编码的转换。甚至，Commons Codec还提供了语音编码的转换。除此之外，Commons Codec还对Java原生的消息摘要算法做了良好的封装，提高了方法的易用性。

4.3.1 获得辅助工具

Commons Codec目前提供的加密组件包的版本是1.4，这是一次重大的改进。作者明显感受到Commons Codec对Base64算法和消息摘要算法的支持便利性，是对上一个版本（1.3版）的一次升华。

我们可以直接登录Commons Codec官网（http://commons.apache.org/codec/），下载最新的组件包（http://commons.apache.org/codec/download_codec.cgi）。组件包分为两种压缩格式：

更多资源请访问稀酷客(www.ckook.com)

tar.gz和zip。我们以zip格式为例，commons-codec-1.4-bin.zip包含了二进制可执行文件、源代码和文档，并且在压缩包中都是以jar格式出现的，如图4-5所示。此外还可以获得纯粹的源代码文件commons-codec-1.4-src.zip。

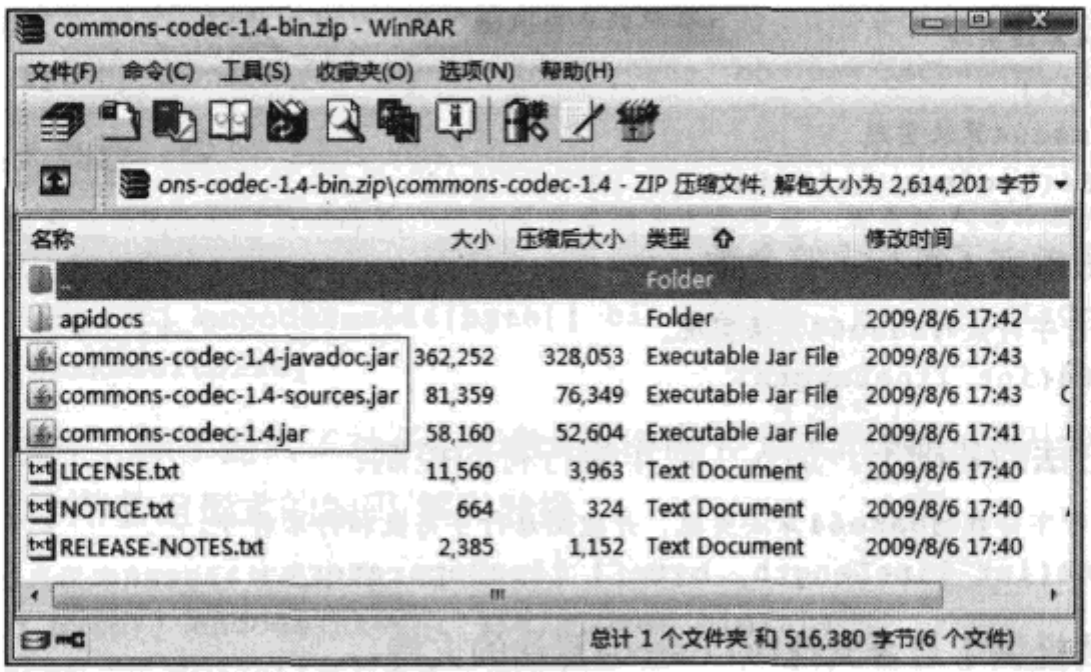


图4-5 commons-codec-1.4-bin.zip文件中包含的相关文件

4.3.2 相关API

Commons Codec的API主要包含了以下几个方面：

- org.apache.commons.codec：该包内主要定义了一些编码转换的接口。
- org.apache.commons.codec.binary：该包内主要完成了编码转换实现，如Base64、二进制、十六进制和字符集编码。
- org.apache.commons.codec.digest：该包内仅有一个实现类DigestUtils，它是对Java原生消息摘要实现的改进。
- org.apache.commons.codec.language：该包内主要完成了语言和语音编码器实现。
- org.apache.commons.codec.net：该包内主要完成了网络相关的编码和解码，如Url编码/解码。

本书将主要阐述用于Base64和十六进制编码的转换实现，以及消息摘要算法相关实现。

1. Base64

Commons Codec也提供了用于Base64算法的实现类，与Bouncy Castle同名——Base64类。与Bouncy Castle提供的Base64类有所不同的是，Commons Codec提供的Base64类遵循RFC 2045（http://www.ietf.org/rfc/rfc2045.txt）。RFC全称为“Request For Comments”，意为请求注解。有关细节，请参见第5章的内容。

同时，这个Base64类实现了Bouncy Castle提供的UrlBase64类的Url编码功能。

```
// 用于Base64编码/解码转换。
public class Base64
implements BinaryEncoder, BinaryDecoder
```

更多资源请访问稀酷客(www.ckook.com)

❑ 方法详述

这个Base64类提供了相关的构造方法，用于控制构建一般Base64算法或Url Base64算法。以下两个构造方法互为映射：

```
// 构建Base64算法实现
public Base64()
// 构建Url Base64算法实现
public Base64(boolean urlSafe)
```

以下构造方法指定了每行字符个数：

```
// 构建指定每行字符数的Base64算法实现。
public Base64(int lineLength)
```

在上述构造方法的基础上，加入了回车换行符的控制：

```
// 构建指定每行字符数的Base64算法实现，并控制每行字符数和行末符号。
public Base64(int lineLength, byte[] lineSeparator)
```

以下构造方法提供了是否支持Url Base64算法的支持：

```
/* 构建指定每行字符数的Base64算法实现，并控制每行字符数和行末符号及是否支持Url Base64算法。*/
public Base64(int lineLength, byte[] lineSeparator, boolean urlSafe)
```

我们先来了解Base64类的静态方法。

下面是最为常用的Base64编码方法：

```
// 以字节数组形式返回Base64编码结果。
public static byte[] encodeBase64(byte[] binaryData)
```

这个方法返回的是一般Base64编码，与之不同的是下面两种方法：

```
/* 以字节数组形式返回Base64编码结果，输出结果中每76个字符追加一个回车换行符。*/
public static byte[] encodeBase64Chunked(byte[] binaryData)
// 以字符串形式返回Base64编码结果，输出结果中每76个字符追加一个回车换行符。
public static String encodeBase64String(byte[] binaryData)
```

在内部实现上，上述方法均调用了以下这种方法：

```
/* 以字节数组形式返回Base64编码结果，对输出结果中每76个字符追加一个回车换行符可控。*/
public static byte[] encodeBase64(byte[] binaryData, boolean isChunked)
```

对应上述编码方法，以下是相应的解码方法：

```
// 以字节数字形式返回Base64解码结果。
public static byte[] decodeBase64(byte[] base64Data)
// 以字节数字形式返回Base64解码结果。
public static byte[] decodeBase64(String base64String)
```

在4.2.3节中，我们提到了Url Base64算法。主要差别是将原Base64字符映射表中的“+”和“/”替换为“-”和“_”。Commons Codec的Base64类也完成了同样的实现，提供了如下方法：

```
// 以字节数组形式返回Url Base64编码结果。
public static byte[] encodeBase64URLSafe(byte[] binaryData)
```

更多资源请访问稀酷客(www.ckook.com)

```
// 以字符串形式返回Url Base64编码结果。
public static String encodeBase64URLSafeString(byte[] binaryData)
```

上述两种方法实际上调用了如下方法：

```
/* 以字节数组形式返回Base64编码结果，是否加入回车换行符可控，是否使用Url Base64编码可控。*/
public static byte[] encodeBase64(byte[] binaryData, boolean isChunked, boolean urlSafe)
```

下述方法则更为细致一些：

```
/* 以字节数组形式返回Base64编码结果，对结果中每行多少个字符、是否加入回车换行符可控，是否使用
Url Base64编码可控。*/
public static byte[] encodeBase64(byte[] binaryData, boolean isChunked, boolean
urlSafe, int maxResultSize)
```

在这个Base64类中，除了提供了对于字节数组和字符串形式的编码/解码转换，同时也提供了BigInteger类型和字节数组形式的编码/解码转换。

```
// 以字节数组形式返回Base64编码结果。
public static byte[] encodeInteger(BigInteger bigInt)
// 以BigInteger形式返回Base64解码结果。
public static BigInteger decodeInteger(byte[] pArray)
```

如果需要判断一个字节数组（或一个字节）中是否包含了Base64字符映射表中的字符，可以使用如下方法：

```
// 测试输入的字节数组是否包含Base64字符映射表中的字符。
public static boolean isArrayByteBase64(byte[] arrayOctet)
// 判别输入字节是否在Base64字符映射表中。
public static boolean isBase64(byte octet)
```

在对上述静态方法了解一番后，下述方法理解起来就相对容易多了。它们绝大部分调用了上述静态方法。

以下方法互为Base64编码/解码实现：

```
// 以字节数组形式返回Base64编码结果。
public byte[] encode(byte[] pArray)
// 以字节数组形式返回Base64解码结果。
public byte[] decode(byte[] pArray)
```

如果需要字节数组和字符串之间的Base64编码/解码实现，可使用如下方法：

```
// 以字节数组形式返回Base64解码结果。
public byte[] decode(String pArray)
// 以字符串形式返回Base64编码结果。
public String encodeToString(byte[] pArray)
```

以下方法用于对象形式的Base64编码/解码操作：

```
// 以对象形式返回Base64编码结果。
public Object encode(Object pObject)
// 以对象形式返回Base64解码结果。
public Object decode(Object pObject)
```

更多资源请访问稀酷客(www.ckook.com)

此外，完成Base64初始化后，你可以通过如下方法获知当前实例化对象是否支持Url Basea64算法。

```
// 判别是否是Url Base64编码。
public boolean isUrlSafe()
```

❑ 实现示例

对应4.2.3节中Base64算法的编码解码实现如代码清单4-4所示。

代码清单4-4 Base64编码/解码2

```
String str = "Base64 编码";
System.err.println("原文:\t" + str);
byte[] input = str.getBytes();
// Base64编码
byte[] data = Base64.encodeBase64(input);
System.err.println("编码后:\t" + new String(data));
// Base64解码
byte[] output = Base64.decodeBase64(data);
System.err.println("解码后:\t" + new String(output));
```

控制台输出与4.2.3节中一致：

```
原文:       Base64 编码
编码后:     QmFzZTY0IOe8lueggQ==
解码后:     Base64 编码
```

对应4.2.3节中Url Base64算法的编码解码实现如下：

```
String str = "Base64 编码";
System.err.println("原文:\t" + str);
byte[] input = str.getBytes();
// Url Base64编码
byte[] data = Base64.encodeBase64URLSafe(input);
System.err.println("编码后:\t" + new String(data));
// Url Base64解码
byte[] output = Base64.decodeBase64(data);
System.err.println("解码后:\t" + new String(output));
```

注意控制台输出结果为：

```
原文:       Base64 编码
编码后:     QmFzZTY0IOe8lueggQ
解码后:     Base64 编码
```

读者需要注意这一点，Bouncy Castle和Commons Codec对于Url Base64算法的理解上有所不同。关于Url Base64算法，本身没有一个统一的标准，没有像RFC 2045这样的明文规定。

两者不同之处在于对原Base64算法中补位概念的理解，如下所示：

❑ Bouncy Castle：使用“.”符号替代“=”进行补位。

❑ Commons Codec：不进行补位。

更多资源请访问稀酷客(www.ckook.com)

在使用Url Base64算法时，需要注意选择合适的实现。

2. Base64InputStream

Commons Codec在1.4版本中，加入了对Base64输入/输出流的支持，包含Base64InputStream和Base64OutputStream两个类。

```
// 完成Base64编码/解码输入流操作。
public class Base64InputStream
extends FilterInputStream
```

❑ 方法详述

既然是流操作，就一定有相应的构造方法。Base64InputStream类可通过如下构造方法获得实例化对象：

```
// 通过输入流构造，默认不支持Base64编码。
public Base64InputStream(InputStream in)
```

如果需要支持Base64编码，需要使用如下方法，将doEncode参数设置为true。

```
// 通过输入流构造，可设置是否支持Base64编码。
public Base64InputStream(InputStream in, boolean doEncode)
```

以下方法还指定了每行字符数及行末符号：

```
// 通过输入流构造，可设置是否支持Base64编码，并指定每行字符数及行末符号。
public Base64InputStream(InputStream in, boolean doEncode, int lineLength,
byte[] lineSeparator)
```

获得输入流最主要的操作就是进行读操作了，Base64InputStream类提供如下两种方法：

```
// 每次读一个字节。
public int read()
// 按偏移量读入字节数组。
public int read(byte[] b, int offset, int len)
```

下述方法在当前版本未实现：

```
// 用于测试是否支持标记和重置操作，未实现返回值为false。
public boolean markSupported()
```

其余FilterInputStream类提供的方法，Base64InputStream类未覆盖。

❑ 实现示例

如果在通信环境中，消息收发双方使用Base64算法对消息隐蔽就可以使用Base64InputStream和Base64OutputStream类了。我们通过代码清单4-5来演示如何接收发送方传递过来的Base64编码消息。

代码清单4-5 Base64输入流操作

```
// 实例化Base64InputStream，用作Base64解码。
Base64InputStream input = new Base64InputStream(is, false);
// 使用DataInputStream包装Base64InputStream。
DataInputStream dis = new DataInputStream(input);
```

更多资源请访问稀酷客(www.ckook.com)

```
// 信息体
byte[] data = new byte[contentLength];
// 读入
dis.readFully(data);
// 关闭流
dis.close();
// 控制台输出
System.err.println(new String(data));
```

其中，is为网络输入流，contentLength为网络流长度。通过上述实现，在控制台中，我们获得如下内容：

```
Base64 解码
```

在网络流完成读取后，获得了解码结果。数据是如何编码并转换为网络流呢？请读者关注下面的内容。

3. Base64OutputStream

Base64OutputStream类自然是对Base64算法输出流的支持。

```
// 完成Base64编码/解码输出流操作。
public class Base64OutputStream
extends FilterOutputStream
```

❑ 方法详述

Base64OutputStream类可通过如下构造方法获得实例化对象：

```
//通过输入流构造，默认不支持Base64编码。
public Base64OutputStream(OutputStream out)
```

如果需要Base64编码需要使用如下构造方法，并将doEncode设置为true：

```
// 通过输入流构造，可设置是否支持Base64编码。
public Base64OutputStream(OutputStream out, boolean doEncode)
```

同时，我们还可以在构造操作时指定每行字符数及行末符号，如下方法所示：

```
// 通过输入流构造，可设置是否支持Base64编码，并指定每行字符数及行末符号。
public Base64OutputStream(OutputStream out, boolean doEncode, int lineLength,
byte[] lineSeparator)
```

既然是输出流，最主要的就是写操作了，也就是如下方法：

```
// 写操作，写入b[]中。
public void write(byte[] b, int offset, int len)
// 写操作，写入b[]中。
public void write(int i)
```

完成写操作后，需要执行如下操作：

```
// 清空流
public void flush()
// 关闭流
public void close()
```

更多资源请访问稀酷客(www.ckook.com)

❑ 实现示例

在消息发送时对输出流做相应实现如代码清单4-6所示。

代码清单4-6 Base64输出流操作

```
// 实例化Base64OutputStream，用作Base64编码。
Base64OutputStream output = new Base64OutputStream(os, true);
// 使用DataOutputStream包装Base64OutputStream。
DataOutputStream dos = new DataOutputStream(output);
// 写操作
dos.write(data);
// 清空
dos.flush();
// 关闭流
dos.close();
```

其中，os是网络输出流，data中是我们传输的消息。

相信读者已经猜到作者传送什么内容了，参见如下代码：

```
String str = "Base64 编码";
byte[] data = str.getBytes();
```

4. Hex

Commons Codec也提供了用于十六进制编码/解码转换的实现类，与Bouncy Castle所提供的类同名，也叫做Hex类。所不同的是，Commons Codec提供了更为细致的API。

```
// 用于十六进制编码/解码转换。
public class Hex
implements BinaryEncoder, BinaryDecoder
```

❑ 方法详述

Hex类与Base64类相类似，既提供静态方法，也提供对应接口实现的方法。以下为该类的构造方法：

```
// 空构造方法，使用默认字符集。
public Hex()
// 根据指定字符集构造。
public Hex(String csName)
```

我们先来了解Hex类的静态方法。

Hex类提供了4种静态方法，以下是编码操作方法：

```
// 以字符数组形式返回十六进制编码结果。
public static char[] encodeHex(byte[] data)
// 以字符串形式返回十六进制编码结果。
public static String encodeHexString(byte[] data)
```

有时候我们对输出的十六进制编码结果有要求，字母必须大写/小写，可以使用如下方法：

```
// 以字符数组形式返回十六进制编码结果，对结果字符大小写可控。
public static char[] encodeHex(byte[] data, boolean toLowerCase)
```

更多资源请访问稀酷客(www.ckook.com)

上述3种方法中，最为常用的非encodeHexString()方法莫属了。

尽管编码方法很多，但解码方法只有这一个：

```
// 以字节数组形式返回十六进制解码结果。
public static byte[] decodeHex(char[] data)
```

以下两种方法，实际上是调用了上述相应的静态方法：

```
// 以字节数组形式返回十六进制编码结果。
public byte[] encode(byte[] array)
// 以字节数组形式返回十六进制解码结果。
public byte[] decode(byte[] array)
```

以下是对对象的编码/解码方法：

```
// 以对象的形式返回对对象十六进制编码结果。
public Object encode(Object object)
// 以对象的形式返回对对象十六进制解码结果。
public Object decode(Object object)
```

对于对象的编码/解码操作，实际上是先判断输入参数是否是String类型，如果不是则强制转换为byte[]类型。如果转换失败，则抛出异常。

此外，Hex类还提供了如下方法：

```
// 获得字符集名称
public String getCharsetName()
// 转换字符串输出
public String toString()
```

❑ 实现示例

参考4.2.3节中十六进制编码/解码实现示例，做相应修改，如代码清单4-7所示。

代码清单4-7 Hex编码/解码2

```
String str = "Hex 编码";
System.err.println("原文:\t" + str);
byte[] input = str.getBytes();
// Hex编码
String data = Hex.encodeHexString(input);
System.err.println("编码后:\t" + data);
// Hex解码
byte[] output = Hex.decodeHex(data.toCharArray());
System.err.println("解码后:\t" + new String(output));
```

观察控制台输出的内容，一定是一样的结果：

```
原文:       Hex 编码
编码后:     48657820e7bc96e7a081
解码后:     Hex 编码
```

5. DigestUtils

DigestUtils类是对Sun提供的MessageDigest类进行了一次封装，提供了更为实用的算法支

更多资源请访问稀酷客(www.ckook.com)

持，弥补了消息摘要结果十六进制编码转换的缺憾。

```
// 用于实现MD5和SHA系列的消息摘要算法。
public class DigestUtils
```

❑ 方法详述

DigestUtils类是一个工具类，它提供了MD5和SHA系列消息摘要算法的实现。

以下是DigestUtils类提供的用于完成MD5算法的静态方法：

```
// 以字节数组形式返回MD5消息摘要信息。
public static byte[] md5(byte[] data)
// 以字节数组形式返回MD5消息摘要信息。
public static byte[] md5(InputStream data)
// 以字节数组形式返回MD5消息摘要信息。
public static byte[] md5(String data)
```

下述方法提供了MD5算法十六进制字符串形式的结果：

```
// 以十六进制字符串形式返回MD5消息摘要信息。
public static String md5Hex(byte[] data)
// 以十六进制字符串形式返回MD5消息摘要信息。
public static String md5Hex(InputStream data)
// 以十六进制字符串形式返回MD5消息摘要信息。
public static String md5Hex(String data)
```

DigestUtils类提供了Java 6所支持的全部SHA系列算法，包括SHA-1、SHA-256、SHA-384和SHA-512算法。

以下是DigestUtils类提供的用于完成SHA-1算法的静态方法：

```
// 以字节数组形式返回SHA消息摘要信息。
public static byte[] sha(byte[] data)
// 以字节数组形式返回SHA消息摘要信息。
public static byte[] sha(InputStream data)
// 以字节数组形式返回SHA消息摘要信息。
public static byte[] sha(String data)
```

下述方法提供了SHA-1算法十六进制字符串形式的结果：

```
// 以十六进制字符串形式返回SHA消息摘要信息。
public static String shaHex(byte[] data)
// 以十六进制字符串形式返回SHA消息摘要信息。
public static String shaHex(InputStream data)
// 以十六进制字符串形式返回SHA消息摘要信息。
public static String shaHex(String data)
```

以下是DigestUtils类提供的用于完成SHA-256算法的静态方法：

```
// 以字节数组形式返回SHA-256消息摘要信息。
public static byte[] sha256(byte[] data)
// 以字节数组形式返回SHA-256消息摘要信息。
public static byte[] sha256(InputStream data)
```

更多资源请访问稀酷客(www.ckook.com)

```
// 以字节数组形式返回SHA-256消息摘要信息。
public static byte[] sha256(String data)
```

下述方法提供了SHA-256算法十六进制字符串形式的结果：

```
// 以十六进制字符串形式返回SHA-256消息摘要信息。
public static String sha256Hex(byte[] data)
// 以十六进制字符串形式返回SHA-256消息摘要信息。
public static String sha256Hex(InputStream data)
// 以十六进制字符串形式返回SHA-256消息摘要信息。
public static String sha256Hex(String data)
```

以下是DigestUtils类提供的用于完成SHA-384算法的静态方法：

```
// 以字节数组形式返回SHA-384消息摘要信息。
public static byte[] sha384(byte[] data)
// 以字节数组形式返回SHA-384消息摘要信息。
public static byte[] sha384(InputStream data)
// 以字节数组形式返回SHA-384消息摘要信息。
public static byte[] sha384(String data)
```

下述方法提供了SHA-384算法十六进制字符串形式的结果：

```
// 以十六进制字符串形式返回SHA-384消息摘要信息。
public static String sha384Hex(byte[] data)
// 以十六进制字符串形式返回SHA-384消息摘要信息。
public static String sha384Hex(InputStream data)
// 以十六进制字符串形式返回SHA-384消息摘要信息。
public static String sha384Hex(String data)
```

以下是DigestUtils类提供的用于完成SHA-512算法的静态方法：

```
// 以字节数组形式返回SHA-512消息摘要信息。
public static byte[] sha512(byte[] data)
// 以字节数组形式返回SHA-512消息摘要信息。
public static byte[] sha512(InputStream data)
// 以字节数组形式返回SHA-512消息摘要信息。
public static byte[] sha512(String data)
```

下述方法提供了SHA-512算法十六进制字符串形式的结果：

```
// 以十六进制字符串形式返回SHA-512消息摘要信息。
public static String sha512Hex(byte[] data)
// 以十六进制字符串形式返回SHA-512消息摘要信息。
public static String sha512Hex(InputStream data)
// 以十六进制字符串形式返回SHA-512消息摘要信息。
public static String sha512Hex(String data)
```

❑ 实现示例

我们以MD5算法摘要处理为例，如代码清单4-8所示。

代码清单4-8 MD5摘要处理

```
String inputStr = "MD5消息摘要";
```

更多资源请访问稀酷客(www.ckook.com)

```
System.err.println("原文: \t" + inputStr);
// 执行MD5消息摘要
String md5Hex = DigestUtils.md5Hex(inputStr);
System.err.println("加密后: \t" + md5Hex);
```

注意控制台的输出：

```
原文:       MD5消息摘要
加密后:     aa24863099cf24696d4eb0f82c918849
```

原文MD5处理后，获得了一个32个字符的十六进制字符串。

有关消息摘要算法相关内容，请参见第6章。

4.4 小结

受限于美国出口限制，我们获得的JDK（或JRE）本身不具备很高的加密强度。在当地的相应进出口政策允许的前提下，我们可以从Sun官方网站上下载相应的权限文件，在一定范围内提高JDK（JRE）的加密强度。

获得Sun官方网站提供的权限文件后，我们只需要在对应的目录下，进行权限覆盖即可。

Bouncy Castle作为一款开源加密组件，极大地丰富了Java世界的加密算法支持。

Sun提供了JCE框架用于支持各种加密算法。基于这个架构，Bouncy Castle提供了相应的安全提供者，扩展了Java 6所不支持的多种算法，如MD4、SHA-224、HmacMD4和HmacSHA-224等消息摘要算法，IDEA对称加密算法以及ECDSA数字签名算法。

使用Bouncy Castle提供者有两种方式：第一种是配置方式，修改JDK（JRE）环境，也就是修改%JDK_Home%\ jre\lib\security\java.security文件（%JRE_Home%\ lib\security\java.security文件），增加Bouncy Castle提供者；第二种是显示调用方式，在使用Bouncy Castle所提供的扩展算法时，先要在各种密钥工厂、密钥对生成器等引擎初始化前，使用Security类的addProvider()方法加入Bouncy Castle提供者。

两种方式各具优势：前者需要依赖环境，但代码十分简洁。开发者必须知道环境对这些扩展算法已支持，无须关注由谁提供。在做迁移时，如项目上线时需要注意相关环境配置；后者需要依赖代码修改，需要使用者十分清楚所需要的扩展算法如何使用及由谁提供。

从架构和开发的角度来考虑：前者属于架构层面，更注重整体；后者注重开发细节，注重单模块。读者需根据开发需求选择合理的实现方式。

对于Base64、十六进制转换算法，Bouncy Castle也提供了相应的支持。此外，Bouncy Castle提供了用于Url编码的UrlBase64类。

Commons Codec是国际开源组织Apache旗下的一款开源组件，提供了各种编码转换实现，如Base64、二进制、十六进制、语音等多种编码转换，并且在Sun提供的消息摘要算法的基础上做了进一步封装，提高了各种算法操作的易用性。

Commons Codec提供的Base64类除了完成一般Base64算法实现，同时实现了Url Base64算法实现。

鱼和熊掌不能兼得，请读者根据相应需求选择Bouncy Castle或Commons Codec。

更多资源请访问稀酷客(www.ckook.com)

更多资源请访问稀酷客(www.ckook.com)

Part2 第二部分

实 践 篇

第5章 电子邮件传输算法——Base64

第6章 验证数据完整性——消息摘要算法

第7章 初等数据加密——对称加密算法

第8章 高等数据加密——非对称加密算法

第9章 带密钥的消息摘要算法——数字签名算法

更多资源请访问稀酷客(www.ckook.com)

第5章

电子邮件传输算法——Base64

Base64是什么？它和加密解密操作有什么关系吗？让我们先来看看以下这段看似诡异的乱码：

```
SmF2YeWKoOWvhuS4juino+WvhueahOiJuuacrw==
```

没错，这就是经过Base64编码后的字符串。对它解码后，我们获得以下内容：

```
Java加密与解密的艺术
```

一段文字在经过Base64编码后面目全非，而经过Base64解码后又能恢复本来面目，这很有加密解密的意味。不过Base64算法并不是加密算法，仅仅是加密算法的近亲。Base64算法的转换方式很像古典加密算法中的单表置换算法。本章将向读者介绍这种常用于电子邮件的算法。

5.1 Base64算法的由来

Base64算法最早应用于解决电子邮件传输的问题。在早期，由于“历史问题”，电子邮件只允许ASCII码字符。如果要传输一封带有非ASCII码字符的电子邮件，当它通过有“历史问题”的网关时就可能出现问题。这个网关很可能会对这个非ASCII码字符的二进制位做调整，即将这个非ASCII码的8位二进制码的最高位置为0。此时用户收到的邮件就会是一封纯粹的乱码邮件了。基于这个原因产生了Base64算法。

5.2 Base64算法的定义

Base64是一种基于64个字符的编码算法，根据RFC 2045（http://www.ietf.org/rfc/rfc2045.txt）的定义：“Base64内容传送编码是一种以任意8位字节序列组合的描述形式，这种形式不易被人直接识别（The Base64 Content-Transfer-Encoding is designed to represent arbitrary sequences of octets in a form that need not be humanly readable.）”。经过Base64编码后的数据会比原始数据略长，为原来的4/3倍。经Base64编码后的字符串的字符数是以4为单位的整数倍。

RFC 2045还规定，在电子邮件中，每行为76个字符，每行末需添加一个回车换行符（“\r\n”），

更多资源请访问稀酷客(www.ckook.com)

不论每行是否够76个字符，都要添加一个回车换行符。但在实际应用中，往往根据实际需要忽略了这一要求。

在RFC 2045文件中给出了如下字符映射表：

Base64字符映射表

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v	(pad)	=
14	O	31	f	48	w		
15	P	32	g	49	x		
16	Q	33	h	50	y		

这张字符映射表中，Value指的是十进制编码，Encoding指的是字符，共映射了64个字符，这也是Base64算法命名的由来。映射表的最后一个字符是等号，它是用来补位的。也难怪有经验的读者朋友一看到字符串末尾有个等号，就会联想到Base64算法了。

其实，Base64算法还有几个同胞兄弟，如Base32和Base16算法。为了能在http请求中以Get方式传递二进制数据，由Base64算法衍生出了Url Base64算法。

Url Base64算法主要是替换了Base64 字符映射表中的第62和63个字符，也就是将“+”和“/”符号替换成了“-”和“_”符号。但对于补位符“=”，一种建议是使用“~”符号，另一种建议是使用“.”符号。其中，由于“~”符号与文件系统冲突，不建议使用；而对于“.”符号，如果出现连续两次，则认为是错误。对于补位符的问题，Bouncy Castle和Commons Codec有差别：Bouncy Castle使用“.”作为补位符，而Commons Codec则完全杜绝使用补位符。

有关Base16、Base32和Url Base64算法的详细内容，读者朋友可以参考RFC 4648 (http://www.ietf.org/rfc/rfc4648.txt)，该文档提交于2006年10月，是一份建议标准（Proposed Standard）。

5.3 Base64算法与加密算法的关系

Base64算法有编码和解码操作可充当加密和解密操作，还有一张字符映射表充当了密钥。本书第2章内容中曾讲到过单表置换算法，Base64算法正是运用了这一思想，将原文经二进制

更多资源请访问稀酷客(www.ckook.com)

转换后与字符映射表相对应，得到“密文”。Base64算法经常用做一个简单的“加密”来保护某些数据。

尽管如此，Base64算法仍不能叫做加密算法。Base64算法公开，这一点与柯克霍夫原则并无违背，但充当密钥的字符映射表公开，直接违反了柯克霍夫原则，而且Base64算法的加密强度并不够高。因此，不能将Base64算法看做我们所认可的现代加密算法。

Base64算法虽不能称为加密算法，但其变换法则遵从了单表置换算法。也正因如此，Base64算法成为加密算法学习最好的范例。尤其是在自定义加密算法研制方面，Base64算法是一个很不错的参考，这也是本书将其纳入的缘由之一。如果我们将Base64算法做少许改造，并将字符映射表调整、保密，改造后的Base64算法就具备了加密算法的意义！除此之外，Base64算法常作为密钥、密文和证书的一种通用存储编码格式，与加密算法形影不离。

5.4 实现原理

Base64算法主要是将给定的字符以与字符编码（如ASCII码，UTF-8码）对应的十进制数为基准，做编码操作：

1）将给定的字符串以字符为单位，转换为对应的字符编码（如ASCII码）。

2）将获得的字符编码转换为二进制码。

3）对获得的二进制码做分组转换操作，每3个8位二进制码为一组，转换为每4个6位二进制码为一组（不足6位时低位补0）。这是一个分组变化的过程，3个8位二进制码和4个6位二进制码的长度都是24位（$3 \times 8 = 4 \times 6 = 24$）。

4）对获得的4-6二进制码补位，向6位二进制码添加2位高位0，组成4个8位二进制码。

5）将获得的4-8二进制码转换为十进制码。

6）将获得的十进制码转换为Base64字符表中对应的字符。

5.4.1 ASCII码字符编码

我们对字符串“A”进行Base64编码，如下所示：

字符	A		
ASCII码	65		
二进制码	01000001		
4-6二进制码	010000	010000	
4-8二进制码	00010000	00010000	
十进制码	16	16	
字符表映射码	Q	Q	= =

由此，字符串“A”经过Base64编码后就得到了“QQ==”这样一个字符串。

Base64解码操作就是编码操作的逆运算，反推上述流程很容易获得原文信息，本文不做详述。

作者有意选择了这样一个字符串作为Base64编码，它经过Base64编码后的字符串末尾带着2

更多资源请访问稀酷客(www.ckook.com)

个等号。很显然，当原文的二进制码长度不足24位，最终转换为十进制码时也不足4项，这时就需要用等号补位。如果原文只有一个字符，那么经过Base64编码后的字符串末尾会有2个等号。

经Base64编码后的字符串最多只会有2个等号，这是因为：

```
余数 = 原文字节数 MOD 3
```

这是一个简单的算术问题，余数的值只可能是0、1或2。余数为0时，则原文字节数恰好是3的倍数，没有等号这个尾巴；余数为1时，则为了让Base64编码后的字符数是4的倍数，要补2个等号；同理，余数为2时，则要补1个等号。所以，通常判别一个字符串是不是Base64编码的第一步操作就是判断这个字符串末尾是不是有等号。

这同时也说明，Base64编码后的字符串是以4个字符为单位，其长度只能是4个字符的整数倍。本文开篇对“Java加密与解密的艺术”这个字符串做了Base64编码后获得了一个长度为40个字符的字符串“SmF2YeWKoOWvhuS4juino+WvhueahOiJuuacrw==”，正好说明了这一点。

5.4.2 非ASCII码字符编码

Base64算法很好地解决了非ASCII码字符的传输问题，譬如中文字符的传输问题。

ASCII码可以表示十进制范围为0～127的字符，对应二进制范围是0000 0000 ～ 0111 1111。ASCII码包括阿拉伯数字、大小写英文字母和一些控制符，但却没有包含双字节编码的字符，如中文字符。因此有了GB2312、GBK和UTF-8等编码。GB2312、GBK用2个字节表示一个汉字，UTF-8编码则用3个字节表示一个汉字。Base64算法实现时，对6位二进制码添加高2位0，恰恰保护了非ASCII码字符在通过有问题的网关时不发生问题。

我们以字符串“密”为例，字符串“密”对应的UTF-8编码就是−27，−81，−122。我们对其做如下Base64编码：

字符	密			
UTF-8码	-27	-81	-122	
二进制码	11100101	10101111	10000110	
4-6二进制码	111001	011010	111110	000110
4-8二进制码	00111001	00011010	00111110	00000110
十进制码	57	26	62	6
字符表映射码	5	a	+	G

字符串“密”经过Base64编码后得到字符串“5a+G”。

当然，如果使用GBK编码就不是这个结果了！字符串“密”对应的GBK编码是−61，−36，经过Base64编码后就是字符串“w9w=”了。

5.5 模型分析

经过一番手工演算，我们已经对Base64算法的内部实现了如指掌。那么，在应用中该如何操作它呢？我们一起通过图5-1来做模型分析。

更多资源请访问稀酷客(www.ckook.com)

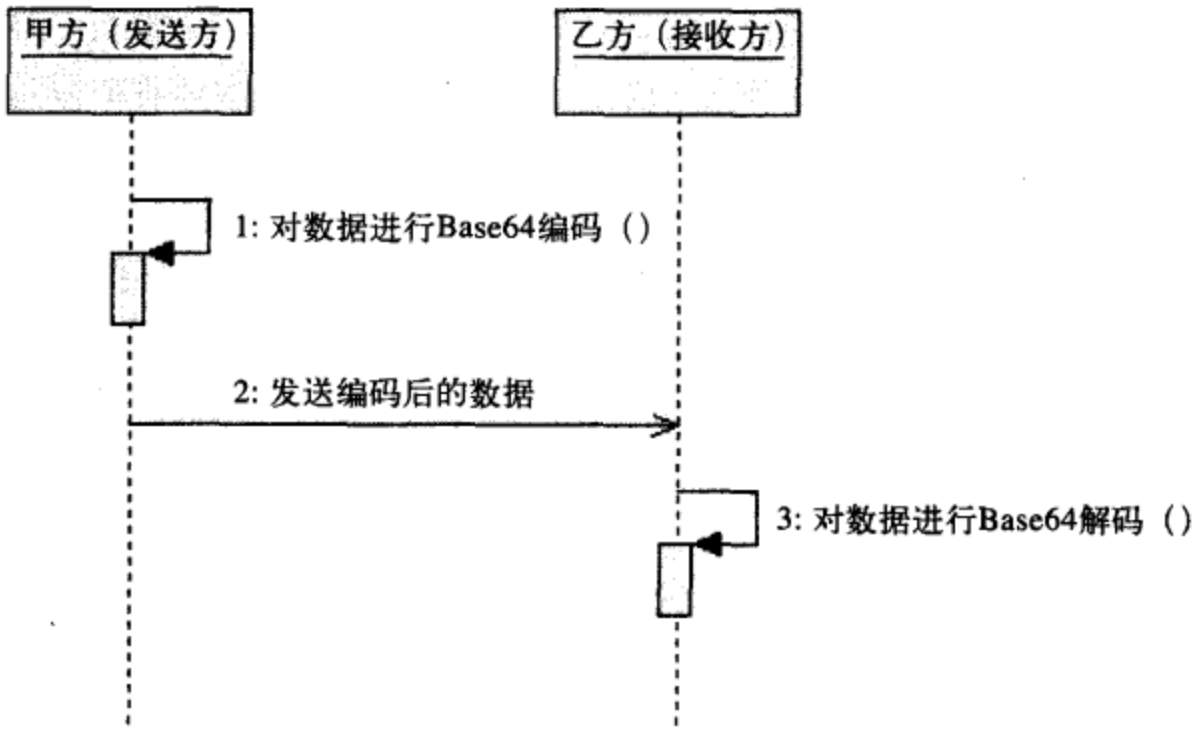


图5-1 基于Base64算法的消息传递模型

我们通过图5-1给出的时序图来描述一次基于Base64算法的消息传递。这里，我们仍以甲方与乙方交互数据举例。甲方作为数据的发送方，乙方作为数据的接收方。其操作流程如下：

1）甲方对数据做Base64编码处理。

2）甲方将编码后的数据发送给乙方。

3）乙方获得数据后对数据做Base64解码处理。

如上所述，要操作Base64算法并不复杂。甲乙双方可以通过上述流程完成一次数据通信；或是单方面地进行电子邮件发送；亦或是仅仅为了传输非ASCII码的字符信息等。这些操作在实际应用中常常是通过Base64算法来完成的。

5.6 Base64算法实现

对于Base64算法，在Java API文档中，我们看不到有关它的任何身影。其实这种算法并不复杂，我们通过手工都能完成编码解码的演算。但是，Sun公司依旧没有提供相应的实现。开源组织总是在必要的时候弥补Sun公司的缺陷，壮大Java的世界。Bouncy Castle和Commons Codec两大开源组件包，填补了Java 6缺失Base64算法实现的遗憾。但实际上Sun也做了Base64算法的实现，却不能将其公示，这令众多Java开发者倍感疑惑。

在Bouncy Castle和Commons Codec两大开源组件包中，Bouncy Castle提供了一般Base64算法的实现，而Commons Codec还提供了基于RFC 2045相关定义的Base64算法实现。

5.6.1 Bouncy Castle

Bouncy Castle遵循的是一般Base64算法，也就是简单地根据Base64字符映射表做了编码转换。对于本文开篇的那一段神奇的乱码转换，我们先通过Bouncy Castle来做一下实现分析。

Bouncy Castle的Base64类对于编码/解码操作只有简单的两个方法，请读者阅读第4章相关

更多资源请访问稀酷客(www.ckook.com)

内容。

我们一起来看代码清单5-1，从而了解用于操作Base64算法的实现类Base64Coder。

代码清单5-1 Base64组件1

```
import org.bouncycastle.util.encoders.Base64;
/**
 * Base64组件
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class Base64Coder {
    // 字符编码
    public final static String ENCODING = "UTF-8";
    /**
     * Base64编码
     * @param data 待编码数据
     * @return String 编码数据
     * @throws Exception
     */
    public static String encode(String data) throws Exception {
            // 执行编码
            byte[] b = Base64.encode(data.getBytes(ENCODING));
            return new String(b, ENCODING);
    }
    /**
     * Base64解码
     * @param data 待解码数据
     * @return String 解码数据
     * @throws Exception
     */
    public static String decode(String data) throws Exception {
            // 执行解码
            byte[] b = Base64.decode(data.getBytes(ENCODING));
            return new String(b, ENCODING);
    }
}
```

Base64算法实现起来就是这么简单，仅用几行代码就完成了编码/解码的调用实现。代码清单5-2是相应的测试用例代码。

代码清单5-2 Base64组件1测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * Base64编码与解码测试类
 * @author 梁栋
```

更多资源请访问稀酷客(www.ckook.com)

```
 * @version 1.0
 * @since 1.0
 */
public class Base64CoderTest {
     // 测试Base64编码与解码
    @Test
    public final void test() throws Exception {
            String inputStr = "Java加密与解密的艺术";
            System.err.println("原文:\t" + inputStr);
            // 进行Base64编码
            String code = Base64Coder.encode(inputStr);
            System.err.println("编码后:\t" + code);
            // 进行Base64解码
            String outputStr = Base64Coder.decode(code);
            System.err.println("解码后:\t" + outputStr);
            // 验证Base64编码解码一致性
            assertEquals(inputStr, outputStr);
    }
}
```

我们终于在控制台中看到了如下信息：

```
原文:       Java加密与解密的艺术
编码后:     SmF2YeWKoOWvhuS4juino+WvhueahOiJuuacrw==
解码后:     Java加密与解密的艺术
```

请大家注意，原文经Base64编码后，并没有多出一个回车换行符！请大家注意图5-2中，code变量的值并没有相应的回车换行符（“\r\n"）。

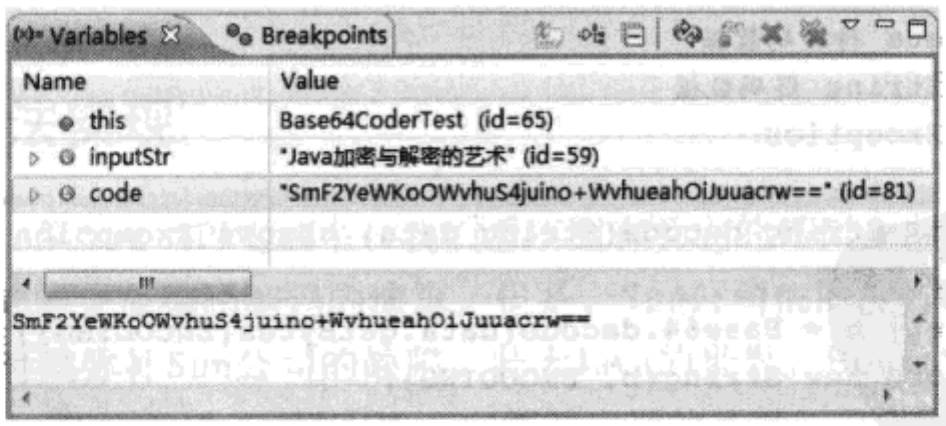


图5-2 Base64 Debug 1

Bouncy Castle未能遵循RFC 2045的相关定义，但在实际使用中，有时候我们恰恰有意需要这么做。如果需要完全遵循RFC 2045，那就使用Commons Codec吧！

5.6.2 Commons Codec

在开源组织中，尤其是Java开源组织中，Apache总能为我们提供各种便利的组件包，使我们的开发工作如虎添翼。Apache提供了用于编码转换的组件包Commons Codec，它遵循RFC 2045的相关定义，实现了Base64算法，同时也支持了一般Base64算法的实现。

更多资源请访问稀酷客(www.ckook.com)

Commons Codec中用于Base64算法的实现是Base64类，请读者朋友查看第4章相关内容。

我们不改变上一小节中对于Base64Coder类的方法定义，只替换对应方法中的代码实现，完成简单的Base64算法实现。

这里请读者朋友注意使用Base64类的静态方法encodeBase64()，它有多种方法重载，可根据参数决定是否执行RFC 2045标准。

如使用以下方法，将完成一般Base64编码：

```
// 以字节数组返回Base64编码结果。
public static byte[] encodeBase64(byte[] binaryData)
```

如果使用以下方法，同时将参数isChunked置为true，将按RFC 2045标准执行：

```
/* 以字节数组形式返回Base64编码结果，对输出结果中每76个字符追加一个回车换行符。*/
public static byte[] encodeBase64(byte[] binaryData, boolean isChunked)
```

接下来我们改用Commons Codec的Base64算法的相应实现，如代码清单5-3所示。

代码清单5-3 Base64组件2

```
import org.apache.commons.codec.binary.Base64;
/**
 * Base64组件
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class Base64Coder {
    // 字符编码
    public final static String ENCODING = "UTF-8";
    /**
     * Base64编码
     * @param data 待编码数据
     * @return String 编码数据
     * @throws Exception
     */
    public static String encode(String data) throws Exception {
            // 执行编码
            byte[] b = Base64.encodeBase64(data.getBytes(ENCODING));
            return new String(b, ENCODING);
    }
    /**
     * Base64安全编码<br>
     * 遵循RFC 2045实现
     * @param data待编码数据
     * @return String 编码数据
     * @throws Exception
     */
    public static String encodeSafe(String data) throws Exception {
```

更多资源请访问稀酷客(www.ckook.com)

```
            // 执行编码
            byte[] b = Base64.encodeBase64(data.getBytes(ENCODING), true);
            return new String(b, ENCODING);
        }
        /**
         * Base64解码
         * @param data 待解码数据
         * @return String 解码数据
         * @throws Exception
         */
        public static String decode(String data) throws Exception {
            // 执行解码
            byte[] b = Base64.decodeBase64(data.getBytes(ENCODING));
            return new String(b, ENCODING);
        }
    }
```

这里对于编码操作实现了2个方法，一个方法用于一般Base64编码实现（encode()方法），另一个用于RFC 2045标准的Base64编码实现（encodeSafe()方法）。

对于解码操作，Commons Codec的Base64类仅需要同一个方法来处理，我们在这里对其包装构造了decode()方法。

对于上述算法实现，给出相应的测试用例，如代码清单5-4所示。

代码清单5-4 Base64组件2测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * Base64编码与解码测试类
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class Base64CoderTest {
    // 测试Base64编码与解码
    @Test
    public final void test() throws Exception {
            String inputStr = "Java加密与解密的艺术";
            System.err.println("原文:\t" + inputStr);
            // 进行Base64编码
            String code = Base64Coder.encode(inputStr);
            System.err.println("编码后:\t" + code);
            // 进行Base64解码
            String outputStr = Base64Coder.decode(code);
            System.err.println("解码后:\t" + outputStr);
            // 验证Base64编码解码一致性
            assertEquals(inputStr, outputStr);
```

更多资源请访问稀酷客(www.ckook.com)

```
    }
    // 测试Base64编码与解码
    @Test
    public final void testSafe() throws Exception {
            String inputStr = "Java加密与解密的艺术";
            System.err.println("原文:\t" + inputStr);
            // 进行Base64编码
            String code = Base64Coder.encodeSafe(inputStr);
            System.err.println("编码后:\t" + code);
            // 进行Base64解码
            String outputStr = Base64Coder.decode(code);
            System.err.println("解码后:\t" + outputStr);
            // 验证Base64编码解码一致性
            assertEquals(inputStr, outputStr);
    }
}
```

我们分别执行两个测试方法。通过执行test()方法，在控制台中获得如下信息：

```
原文:       Java加密与解密的艺术
编码后:     SmF2YeWKoOWvhuS4juino+WvhueahOiJuuacrw==
解码后:     Java加密与解密的艺术
```

观察Debug信息，如图5-3所示。

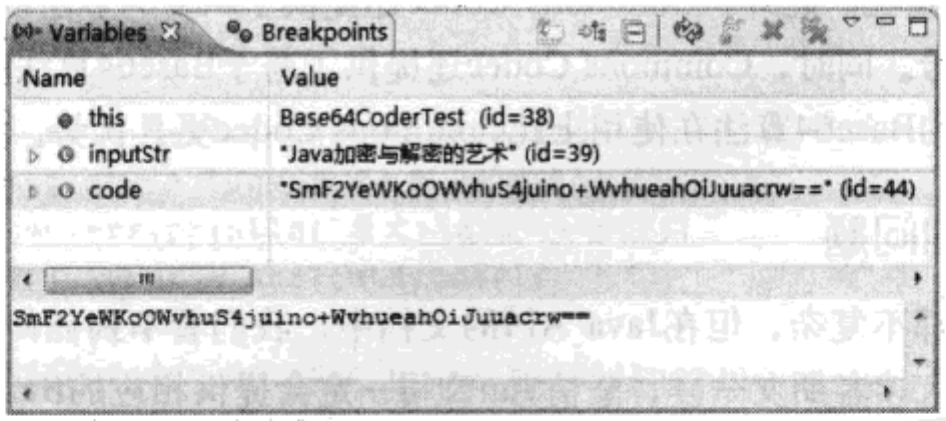


图5-3 Base64 Debug 2

注意图中字符串变量code的值，我们确定Base64Coder类的encode()方法仅实现了一般Base64编码操作。

我们再来执行testSafe()方法，在控制台中获得如下信息：

```
原文:       Java加密与解密的艺术
编码后:     SmF2YeWKoOWvhuS4juino+WvhueahOiJuuacrw==
解码后:     Java加密与解密的艺术
```

这里有个换行，我们很难断定是否还有回车换行符。通过图5-4，我们得到了验证。

注意图中字符串变量code的值，我们明显看到这里有一个回车换行符（“\r\n”）。我们确定Base64Coder类的encodeSafe()方法遵照RFC 2045定义实现了Base64编码操作。

更多资源请访问稀酷客(www.ckook.com)

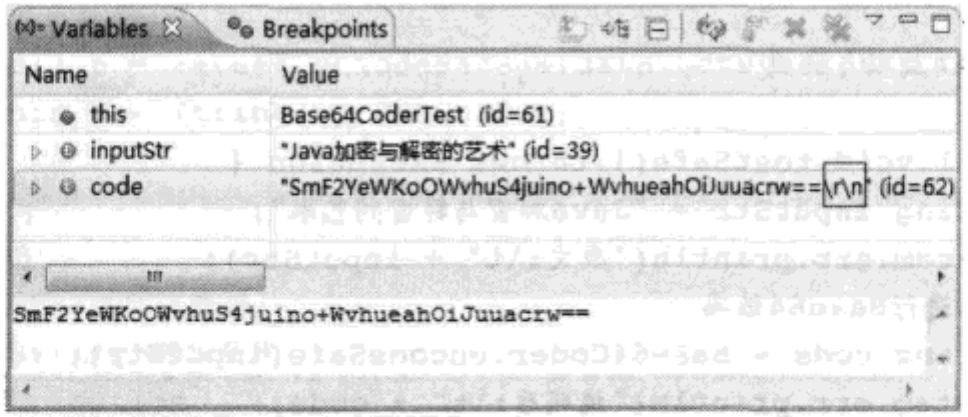


图5-4 Base64 Debug 3

注意 根据RFC 2045定义，每行为76个字符，行末加入一个回车换行符。但并不是当每行够了76个字符才需要在行末加入一个回车换行符，每行不管够不够76个字符都要加入一个回车换行符。

5.6.3 两种实现方式的差异

Bouncy Castle和Commons Codec都提供了Base64算法实现，但是否遵循RFC 2045定义，即在编码后的字符串末尾添加回车换行符，则是两种实现的唯一差别！

Bouncy Castle遵循了一般Base64算法编码。

Commons Codec提供了Base64算法的两种实现标准：一种是遵循一般Base64算法实现；另一种是遵循RFC 2045定义。此外，Commons Codec还提供了Base64算法的定制实现，可以自定每行字符数和行末符号。同时，Commons Codec还提供了基于Base64算法的输入输出流实现。

综上所述，单纯的Base64算法在使用上比Commons Codec更具优势。

5.6.4 不得不说的问题

Base64算法一点都不复杂，但在Java API的文档中，我们看不到任何有关Base64算法的影子。但是，作者和广大读者朋友一样，坚信Sun公司一定会提供相应的Base64算法实现。

在sun.misc包中有两个类，BASE64Encoder类和BASE64Decoder类。BASE64Encoder类用于Base64编码，BASE64Decoder类用于Base64解码。但作者坚决反对使用这些Base64算法的实现类。

1. Base64算法的非标准实现

BASE64Encoder类继承于CharacterEncoder类，用于Base64编码：

```
// 用于Base64编码。
public class BASE64Encoder
extends CharacterEncoder
```

BASE64Encoder类有一个最常用的方法：

```
// 获得给定字节数组的Base64编码字符串。
public String encodeBuffer(byte b[])
```

与BASE64Encoder类相对应，BASE64Decoder类继承于CharacterDecoder类，用于Base64

更多资源请访问稀酷客(www.ckook.com)

解码：

```
// 用于Base64解码。
public class BASE64Decoder
extends CharacterDecoder
```

BASE64Decoder类有一个最常用的方法：

```
// 获得给定的字符串的Base64解码字节数组。
public byte[] decodeBuffer(String s)
```

我们通过代码清单5-5展示如何使用Sun提供的Base64算法实现。

代码清单5-5 Base64算法Sun实现

```
import static org.junit.Assert.*;
import org.junit.Test;
import sun.misc.BASE64Decoder;
import sun.misc.BASE64Encoder;
/**
 * Base64编码与解码测试类
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class Base64CoderTest {
    // 测试Base64编码与解码
    @Test
    public final void test() throws Exception {
            String str = "Java加密与解密的艺术";
            System.err.println("原文:\n\t" + str);
            byte[] input = str.getBytes();
            // Base64编码
            BASE64Encoder encoder = new BASE64Encoder();
            String data = encoder.encodeBuffer(input);
            System.err.println("编码后:\n\t" + data);
            // Base64解码
            BASE64Decoder decoder = new BASE64Decoder();
            byte[] output = decoder.decodeBuffer(data);
            System.err.println("解码后:\n\t" + new String(output));
    }
}
```

以下为上述测试代码的输出结果：

```
原文:
        Java加密与解密的艺术
编码后:
        SmF2YeWKoOWvhuS4juino+WvhueahOiJuuacrw==
解码后:
        Java加密与解密的艺术
```

注意上述输出结果，经Base64编码后的字符串有一个回车换行符（“\r\n”）。

注意观察图5-5中字符串变量code的值。

更多资源请访问稀酷客(www.ckook.com)

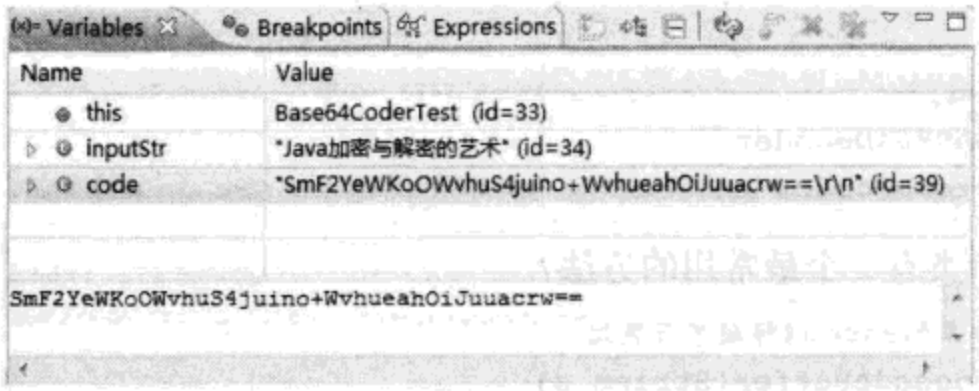


图5-5 Base64 Debug 4

2. 非标准实现的问题

sun.misc包是Sun公司提供给内部使用的专用 API，在编译上述代码时，尤其是使用Ant工具编译代码时，很可能获得以下提示内容：

```
警告: sun.misc.BASE64Decoder是Sun的专用API，可能会在未来版本中删除
import sun.misc.BASE64Decoder;
       ^
警告: sun.misc.BASE64Encoder是Sun的专用API，可能会在未来版本中删除
import sun.misc.BASE64Encoder;
       ^
```

在JRE中，以sun和com.sun开头的包的类都是未被文档化的，它们属于Java和Javax类库的基础，其中的实现大多数与底层平台有关，是不推荐使用的。

从实用角度来讲，我们不能直接使用JDK中以sun开头的包。因为，这是非标准的做法，将带来安全隐患！很有可能如编译器所提示的警告："Sun的专用API，可能会在未来版本中删除"！

如何避免这种问题的出现呢？除了有意识地杜绝使用JDK API以外的JDK底层实现外，我们可以通过Eclipse设置来及时提醒，甚至中止编译。

在Eclipse中设置项目的JRE，如图5-6所示。

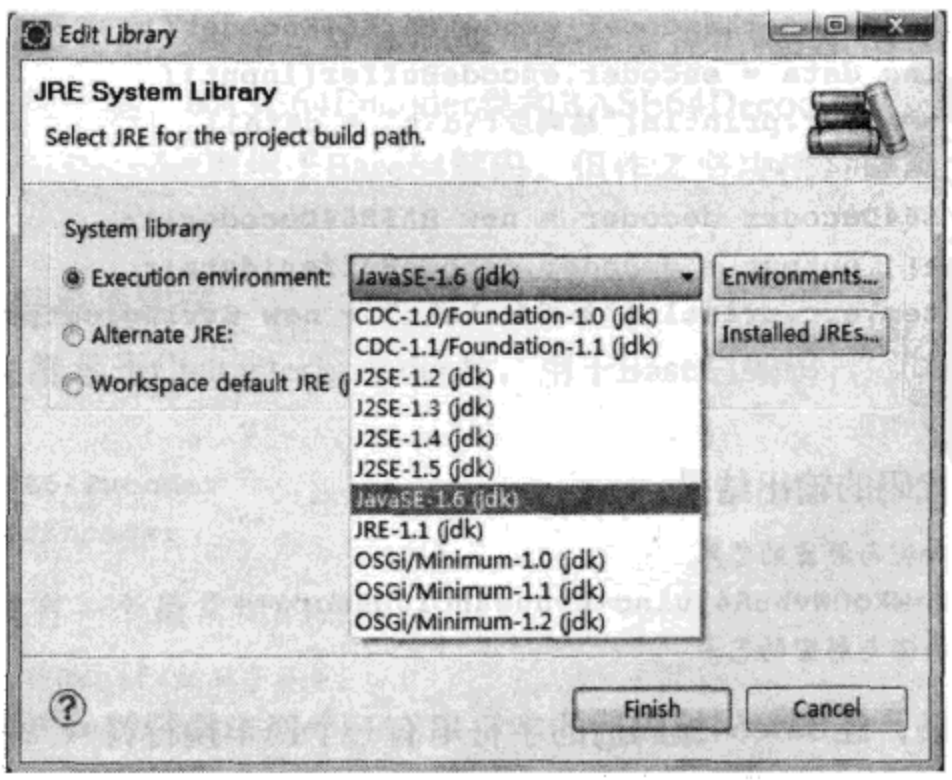


图5-6 在Eclipse中设置项目的JRE

更多资源请访问稀酷客(www.ckook.com)

选择JavaSE-1.6(jdk)，在编译代码时如果引入了非标准的类库，就会有警告，并且不能完成编译，如图5-7所示。

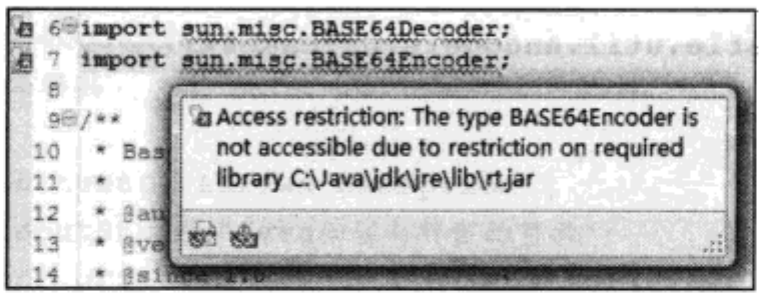


图5-7 在Eclipse中使用非标准类库

正因为Sun提供的Base64算法实现是不推荐使用的，才会有Bouncy Castle和Commons Codec提供相应的解决方案。

作者强烈建议读者朋友（尤其是架构师们），按照标准路线构建应用，避免使用非标准实现，规避潜在风险！

综上所述，如果要使用Base64算法实现，那么Apache的Common Codec是我们最好的选择！

5.7 Url Base64算法实现

相信读者对于URL传递参数已经很熟悉了，如何通过HTTP请求传递URL参数似乎是一个很简单的问题，但问题往往出在非常时期。Get方式和Post方式是最为常用的两种HTTP请求方式，但如果要求使用Get方式传递本该由Post方式传递的私密数据时该怎么办？如果非要在Get方式下传递二进制数据怎么办？此时，使用Url Base64算法是一个不错的选择。

目前，对于Url Base64算法还没有一个明确的定义，我们所获得的最新的相关文档是RFC 4648，其中列举了关于Url Base64算法的构建建议。

Url Base64算法定义有别于RFC 2045，它不需要定义每行字符数及行末回车换行符。同时，根据URL相关要求，符号“+”和符号“/”是不允许出现在URL中的，RFC 4648中给出了相应的替代符号“-”和“_”。同样，符号“=”用做参数分隔符，也不允许出现在URL中。符号“=”在Base64算法中用做填充符，如果需要定长的Base64编码串，就需要有相应的替代符号。

根据RFC 4648中的建议，“~”和“.”符都有可能替代“=”符号。但“~”符号与文件系统相冲突，不能使用；如果使用“.”符号，某些文件系统认为该符号连续出现两次则为错误。正如本章前文所述，Base64编码后的字符串中，填充符可以重复出现，并且最多只能出现两次。如果使用“.”符号作为填充符，则与某些系统相冲突。

Bouncy Castle和Commons Codec都实现了Url Base64算法，不同的是，Bouncy Castle使用“.”作为填充符，而Commons Codec则直接放弃了填充符，使用不定长Url Base64编码。

5.7.1 Bouncy Castle

Bouncy Castle提供了Url Base64算法的实现类UrlBase64，它遵从了RFC 4648中的建议，使

更多资源请访问稀酷客(www.ckook.com)

用“.”符号作为填充符，构建定长Url Base64编码，如代码清单5-6所示。

代码清单5-6 Url Base64处理1

```
import org.bouncycastle.util.encoders.UrlBase64;
/**
 * Url Base64组件
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class UrlBase64Coder {
    // 字符编码
    public final static String ENCODING = "UTF-8";
    /**
     * Url Base64编码
     * @param data待编码数据
     * @return String 编码数据
     * @throws Exception
     */
    public static String encode(String data) throws Exception {
            // 执行编码
            byte[] b = UrlBase64.encode(data.getBytes(ENCODING));
            return new String(b, ENCODING);
    }
    /**
     * Url Base64解码
     * @param data待解码数据
     * @return String 解码数据
     * @throws Exception
     */
    public static String decode(String data) throws Exception {
            // 执行解码
        byte[] b = UrlBase64.decode(data.getBytes(ENCODING));
        return new String(b, ENCODING);
    }
}
```

我们通过代码清单5-7来验证上述操作。

代码清单5-7 Url Base64处理1测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * Url Base64编码与解码测试类
 * @author 梁栋
```

更多资源请访问稀酷客(www.ckook.com)

```
 * @version 1.0
 * @since 1.0
 */
public class UrlBase64CoderTest {
    // 测试Base64编码与解码
    @Test
    public final void test() throws Exception {
            String inputStr = "Java加密与解密的艺术";
            System.err.println("原文:\t" + inputStr);
            // 进行Base64编码
            String code = UrlBase64Coder.encode(inputStr);
            System.err.println("编码后:\t" + code);
            // 进行Base64解码
            String outputStr = UrlBase64Coder.decode(code);
            System.err.println("解码后:\t" + outputStr);
            // 验证Base64编码解码一致性
            assertEquals(inputStr, outputStr);
    }
}
```

注意控制台输出信息：

```
原文:       Java加密与解密的艺术
编码后:     SmF2YeWKoOWvhuS4juino-WvhueahOiJuuacrw..
解码后:     Java加密与解密的艺术
```

我们使用Base64编码后得到的是以下信息：

```
原文:       Java加密与解密的艺术
编码后:     SmF2YeWKoOWvhuS4juino+WvhueahOiJuuacrw==
解码后:     Java加密与解密的艺术
```

对比差别，发现使用Base64编码后的字符串中的“+”和“=”符号分别替换为“-”和“.”符号，而且没有换行的概念。

5.7.2 Commons Codec

Commons Codec在其1.4版本中对原有Base64类的实现做了扩充，增加了Url Base64算法支持，它舍弃了填充符，使用不定长Url Base64编码。

我们来看代码清单5-8如何实现Url Base64编码。

代码清单5-8 Url Base64处理2

```
import org.apache.commons.codec.binary.Base64;
/**
 * Url Base64组件
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
```

更多资源请访问稀酷客(www.ckook.com)

```
public abstract class UrlBase64Coder {
    // 字符编码
    public final static String ENCODING = "UTF-8";
    /**
     * Url Base64编码
     * @param data待编码数据
     * @return String 编码数据
     * @throws Exception
     */
    public static String encode(String data) throws Exception {
            // 执行编码
            byte[] b =  Base64.encodeBase64URLSafe(data.getBytes(ENCODING));
            return new String(b, ENCODING);
    }
    /**
     * Url Base64解码
     * @param data待解码数据
     * @return String 解码数据
     * @throws Exception
     */
    public static String decode(String data) throws Exception {
            // 执行解码
            byte[] b = Base64.decodeBase64(data.getBytes(ENCODING));
            return new String(b, ENCODING);
    }
```

测试用例与代码清单5-7所示内容一致，作者就不在这里复述了。

注意控制台输出的信息：

```
原文：      Java加密与解密的艺术
编码后：    SmF2YeWKoOWvhuS4juino-WvhueahOiJuuacrw
解码后：    Java加密与解密的艺术
```

对比Bouncy Castle的Url Base64编码后的信息：

```
原文：      Java加密与解密的艺术
编码后：    SmF2YeWKoOWvhuS4juino-WvhueahOiJuuacrw..
解码后：    Java加密与解密的艺术
```

唯一的差别在于是否使用“.”符号作为填充符。显然，Commons Codec使用了不定长Url Base64编码。

5.7.3 两种实现方式的差异

两种实现方式的主要差异在于实现标准。

Bouncy Castle的Url Base64算法实现与RFC 4648定义较为接近，构建定长Base64编码，使用“.”符号作为填充符。

更多资源请访问稀酷客(www.ckook.com)

Commons Codec的Url Base64算法实现遵照了RFC 4648绝大部分定义，为避免可能的错误，使用不定长Base64编码，抛弃了填充符。

对于这两种实现，从实用性的角度讲，通过URL参数方式传输数据往往要求数据长度尽量缩短，以缩短URL长度，避免网关限制，减少网络传输时间。从这一点讲，Commons Codec的Url Base64实现较为实用。

5.8 应用举例

Base64算法广泛应用于电子邮件传输，以及密钥和证书文件的文本方式保存。在数据保密要求强度不高的情况下，可以使用Base64算法做简单的数据“加密”。

5.8.1 电子邮件传输

电子邮件一般都会使用Base64算法，我们截取一段电子邮件信息：

```
Content-Type: text/plain;
charset="utf8"
Content-Transfer-Encoding: base64
6L+Z5piv5LiA5bCB5rWL6K+V6YKu5Lu277yB
```

在这封邮件里，我们看到了我们需要的信息：

- 字符集编码：charset charset="utf8"
- 内容传输编码：Content-Transfer-Encoding: base64
- 邮件内容：6L+Z5piv5LiA5bCB5rWL6K+V6YKu5Lu277yB

通过上述信息，我们确定无疑这是一封使用了Base64编码的邮件，并且邮件的内容使用的是UTF-8编码的字符集。由此，我们只需要通过上述Base64算法的实现类——Base64Coder完成解码操作。毋庸置疑，这份邮件的内容是：“这是一封测试邮件！”

5.8.2 网络数据传输

不论是通过HTTP的Get方式以URL参数传输数据，还是通过Post方式以数据体传输数据，都能发现Base64编码藏匿其中。

只要通信双方在通信前约定好字符编码和字符映射表，改良后的Base64算法就可以在HTTP环境中以Get/Post方式传递二进制数据。这时的Base64算法，无疑成为了一种简单的加密算法。

由于RFC 4648还未形成最终标准，在URL中使用Url Base64算法对数据编码/解码还未形成一定规模，因此大部分应用软件都是使用自定义的Url Base64编码格式。

如果Url Base64算法形成最终标准，相信我们一定会在很多应用领域看到它的影子。

更多资源请访问稀酷客(www.ckook.com)

5.8.3 密钥存储

在计算机的世界里，密钥就是一段二进制的数据。我们可以通过Key接口的getEncoded()方法获得密钥对应的字节数组（详见3.2.6节）。

密钥的二进制表现形式使得它的安全性大为提升，但却大大降低了它的可读性。通常我们希望密钥可以有一个容易理解的表现形式（如经过Base64编码后的字符串），以增强它的可读性并且方便密钥的发放。如甲方向乙方发送密钥，可以将密钥以二进制形式转化为Base64编码后的字符串形式，通过安全途径以文档形式发放给乙方，密钥很可能被要求写在合同中。

Base64算法的优势恰恰是在基于二进制编码格式的转换上，这一点使得二进制的密钥通常以Base64编码的形式展现。另一种类似的效果是将二进制串转为十六进制串，如消息摘要结果。

以下是一个经过Base64编码处理后DES的密钥：

```
I6ttW0mtSo8=
```

这样处理后的密钥看起来就比较容易接受，可以将其放到合同书中，发放给合作伙伴！

我们将在本书的后续内容中多次看到上述Base64编码形式的密钥。

5.8.4 数字证书存储

你能想象数字证书（见图5-8）与Base64算法有什么关系吗？借用电影《黑客帝国I》中的经典台词："让我们来看一看真实的世界！"

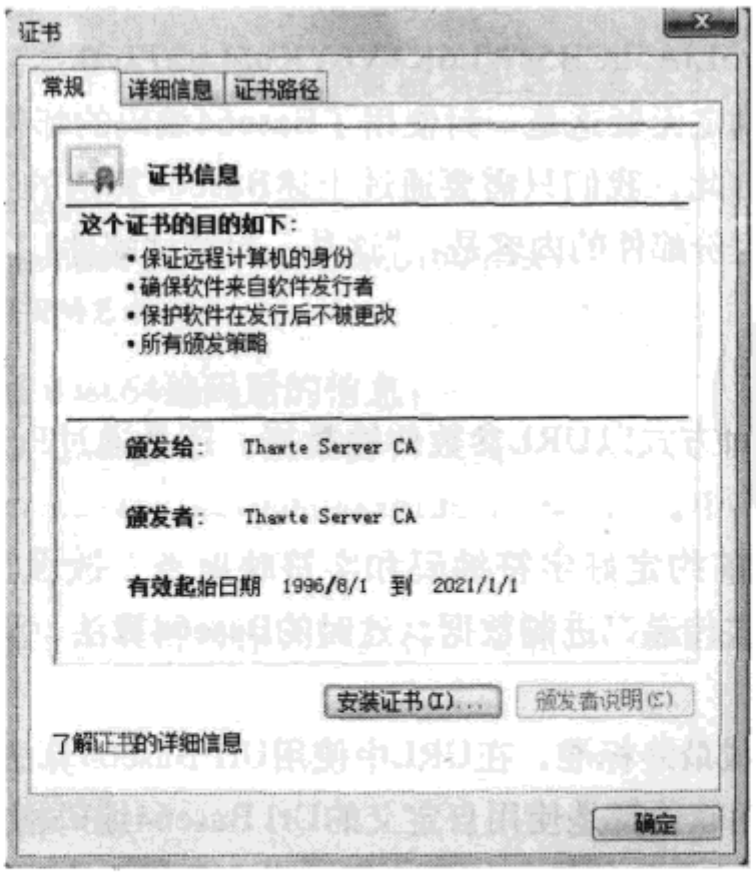


图5-8 数字证书

数字证书可以有多种存储方式，最常用的是DER编码二进制、Base64编码和加密消息语法

更多资源请访问稀酷客(www.ckook.com)

标准等。数字证书Base64编码形式如图5-9所示。它遵从RFC 2045对电子邮件的Base64编码格式要求，每间隔76个字符添加一个回车换行符。在HTTP网络中，数字证书就是以这种编码形式传输的。



图5-9 数字证书Base64编码形式

我们将在本书后续内容中多次看到上述Base64编码形式的数字证书。

5.9 小结

在这一章中，我们了解了Base64算法，它是电子邮件常用的传输算法。电子邮件只允许使用ASCII码，而对于非ASCII码（如中文）就显得无能为力了。因此，为了能在电子邮件中使用非ASCII码，就有了Base64算法。

与我们熟知的加密算法相比，Base64算法算不上真正的加密算法，尽管Base64算法有编码和解码操作可充当加密和解密操作，还有一张字符映射表充当了密钥。它与古典加密算法中的单表置换算法极为相似。但是，Base64算法公开密钥的方式却违反了柯克霍夫原则，而且它的加密强度并不高。因此，Base64算法算不上真正的加密算法。尽管如此，经过改良的Base64算法常常作为加密算法使用。

Bouncy Castle和Commons Codec都提供了Base64算法实现，两组织在算法实现上遵循了不同的标准。Bouncy Castle实现了一般Base64编码，而Commons Codec遵循RFC 2045的相关定义做了算法实现。简而言之，RFC 2045要求Base64编码后的字符串每行76个字符，不论每行是否够76个字符，都要在行末添加一个回车换行符（“\r\n”）。Bouncy Castle并没有对于Base64编码字符串的行要求，而Commons Codec遵循了这些要求并提供了一般Base64算法实现的方法，而且在方法易用性上提供了更为广泛的API，方便使用。因此，对于Base64算法的实现，Commons Codec更胜一筹。

更多资源请访问稀酷客(www.ckook.com)

Sun公司提供了Base64算法的非标准实现。它位于sun.misc包中，提供BASE64Encoder类和BASE64Decoder类，分别用来做Base64算法的编码和解码工作。这两个Base64算法的实现类遵循了RFC 2045的要求。在Java API文档中，我们看不到它们的相关介绍。这是因为它们位于以sun开头的包中，是非标准的实现，是不提倡使用的。

Url Base64算法是Base64算法的一种变体。对于Url Base64算法，目前还没有一个明确的定义，唯一一个可以参考的文档是RFC 4648，其中定义了Url Base64算法的相关要求，调整了原Base64字符映射表，使用“-”和“_”符号分别替换“+”和“/”符号。同时，还建议使用“~”或“.”符号替换“=”符号，用做填充符，但由于这两个候补填充符均存在某些问题，目前还没有定性。

Bouncy Castle和Commons Codec都实现了Url Base64算法。有所不同的是，Bouncy Castle使用“.”作为填充符，而Commons Codec则彻底抛弃了填充符。从实用的角度讲，Commons Codec更具优势。

Base64算法广泛用于电子邮件传输、网络数据传输、密钥存储和数字证书存储。经过改良的Base64算法常作为网络数据传输二进制数据，甚至可以作为一种简单的加密手段。

更多资源请访问稀酷客(www.ckook.com)

第6章

验证数据完整性——消息摘要算法

相信读者对于MySQL已经非常了解了，作为一款开源的数据库，它已经成为无数开源爱好者的数据库首选。

读者朋友一定对图6-1并不陌生，它是MySQL官方提供的数据库发行版（版本5.1.38）的下载页面（详见http://dev.mysql.com/downloads/mysql/5.1.html#win32）。

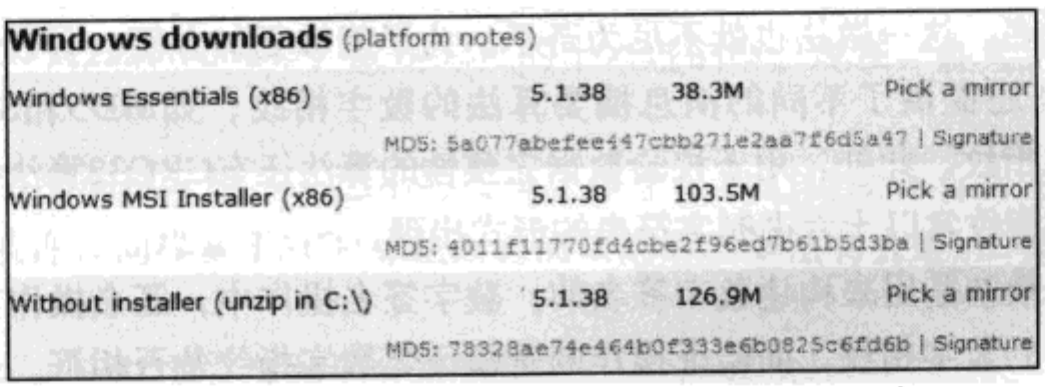


图6-1 MySQL下载页面

在这个下载页面中，有一个不规律的字符串（MD5: 5a077abefee447cbb271e2aa7f6d5a47）引起了作者的注意。这个不规律的字符串长度为32个字符，由英文字母和数字组成，很显然这是一个十六进制编码字符串。它还有一个时髦的名字，叫“数字指纹”。它就是本章的主角，消息摘要算法——MD5。

我们看到Windows Essentials (x86)、Windows MSI Installer (x86)和Without installer (unzip in C:\)三种发行版的大小是有所不同的，三种发行版对应的大小分别为38.3MB、103.5MB和126.8MB。但是，其数字指纹的长度却都是一样的，都是32个字符的十六进制串。

6.1 消息摘要算法简述

消息摘要算法包含MD、SHA和MAC共3大系列，常用于验证数据的完整性，是数字签名算法的核心算法。

6.1.1 消息摘要算法的由来

相信读者朋友都有从网上下载软件的经历，偶尔也有从网上下载到破损文件的经历。情况

更多资源请访问稀酷客(www.ckook.com)

严重时，还可能从某软件的官网上下载到被篡改的软件。如何来验证下载到的文件和官方提供的文件是否一致？这就引入了数据完整性验证的问题。

该如何验证其一致性？肉眼比较？大小比较？均无可取之处！我们需要一种方便快捷、安全有效的算法。

先不说如何比较文件是否相同的问题，我们说说如何比较两个对象是否相同。

相信广大读者朋友都有使用equals()方法来比较对象的经历。但很多读者朋友不知道，实际上equals()方法比较的是两个对象的散列值，即比较两个对象hashCode ()方法的值是否相同，这说明hashCode可以作为辨别对象的唯一标识。

什么是hashCode呢？顾名思义，hashCode就是散列值。

我们在第2章中曾经介绍过散列函数，它恰恰能够用于数据完整性的校验。任何消息经过散列函数处理后，都会获得唯一的散列值。这一过程称为“消息摘要”，其散列值称为“数字指纹”，自然其算法就是“消息摘要算法”了。换句话说，如果其数字指纹唯一，就说明其消息是一致的。

由此，消息摘要算法成了校验数据完整性的主要手段。各大软件厂商提供软件下载的同时总要附带上数字签名，这一做法也就不足为奇了。为了能够更加方便、有效地验证数据的完整性，有的软件厂商还提供了不同的消息摘要算法的数字指纹，如MD5和SHA算法，甚至是HMAC算法的数字指纹。此外，用于校验数据完整性的算法还有CRC32算法等。为了方便人们识别和阅读，数字指纹常以十六进制字符串的形式出现。

消息摘要算法最初是用来构建数字签名的。数字签名操作中，签名操作其实是变相地使用消息摘要算法获得的数字指纹，而验证操作则是验证其数字指纹是否相符。这也是为什么当山东大学王小云教授使用碰撞算法破解了MD5和SHA算法后，使得数字签名在理论上被伪造成为可能。

消息摘要算法一直是非对称加密算法中一项举足轻重的关键性算法。

6.1.2 消息摘要算法的家谱

消息摘要算法又称为散列算法，其核心在于散列函数的单向性。即通过散列函数可获得对应的散列值，但不可通过该散列值反推其原始信息。这是消息摘要算法的安全性的根本所在。

消息摘要算法主要分为三大类：MD（Message Digest，消息摘要算法）、SHA（Secure Hash Algorithm，安全散列算法）和MAC（Message Authentication Code，消息认证码算法）。

如前文所述，MD5、SHA和HMAC都属于消息摘要算法，它们是三大消息摘要算法的主要代表。MD系列算法包括MD2、MD4和MD5共3种算法；SHA算法主要包括其代表算法SHA-1和SHA-1算法的变种SHA-2系列算法（包含SHA-224、SHA-256、SHA-384和SHA-512）；MAC算法综合了上述两种算法，主要包括HmacMD5、HmacSHA1、HmacSHA256、HmacSHA384和HmacSHA512算法。

我们知道，科学技术的发展是不以人的意志为转移的。尽管上述内容列举了各种消息摘要算法，但仍不能满足应用需要。基于这些消息摘要算法，又衍生出了RipeMD系列（包含

更多资源请访问稀酷客(www.ckook.com)

RipeMD128、RipeMD160、RipeMD256、RipeMD320）、Tiger、GOST3411和Whirlpool算法。

6.2 MD算法家族

每当人们一提到消息摘要算法，就很自然地会联想到MD5和SHA算法。MD5算法是典型的消息摘要算法，是计算机广泛使用的杂凑算法之一（又译摘要算法、散列算法），更是消息摘要算法的首要代表。

6.2.1 简述

MD5算法是典型的消息摘要算法，其前身有MD2、MD3和MD4算法，它由MD4、MD3、MD2算法改进而来。不论是哪一种MD算法，它们都需要获得一个随机长度的信息并产生一个128位的信息摘要。如果将这个128位的二进制摘要信息换算成十六进制，可以得到一个32位（每4位二进制数转换为1位十六进制数）的字符串，故我们见到的大部分MD5算法的数字指纹都是32位十六进制的字符串，如本文开篇中，MySQL下载页上的数字指纹（MD5: 5a077abefee447cbb271e2aa7f6d5a47）就是32位的十六进制串。现在，各大主流计算机语言均支持MD5算法。

虽然，MD5算法漏洞越来越多，已不再安全，但至今我们仍没有看到它的下一版本——MD6算法的出现。或许，同样基于MD4算法改进而来的SHA算法将会是MD系列算法的主要替代者。

让我们一同回顾一下MD算法家族的发展历史。

❑ MD2算法

1989年，著名的非对称算法RSA发明人之一——麻省理工学院教授罗纳德·李维斯特（Ronald L. Rivest）开发了MD2算法。这个算法首先对信息进行数据补位，使信息的字节长度是16的倍数。再以一个16位的检验和作为补充信息追加到原信息的末尾。最后根据这个新产生的信息计算出一个128位的散列值，MD2算法由此诞生。

有关MD2算法详情请参见RFC 1319（http://www.ietf.org/rfc/rfc1319.txt）。

❑ MD4算法

1990年，罗纳德·李维斯特教授开发出较之MD2算法有着更高安全性的MD4算法。在这个算法中，我们仍需对信息进行数据补位。不同的是，这种补位使其信息的字节长度加上448个字节后能成为512的倍数（信息字节长度mod 512 = 448）。此外，关于MD4算法的处理与MD2又有很大差别。但最终仍旧是会获得一个128位的散列值。MD4算法对后续消息摘要算法起到了推动作用，许多比较有名的消息摘要算法都是在MD4算法的基础上发展而来的，如MD5、SHA-1、RIPE-MD和HAVAL算法等。

有关MD4算法的详情请参见RFC 1320（http://www.ietf.org/rfc/rfc1320.txt）。

著名开源P2P（Peer-To-Peer，点对点）下载软件EMule（http://www.emule.com）所使用的消息摘要算法正是经过改良后的MD4算法。该算法用于对文件分块后做消息摘要，以验证其文

更多资源请访问稀酷客(www.ckook.com)

件的完整性。

❑ MD5算法

1991年，继MD4算法后，罗纳德·李维斯特教授开发了MD5算法，将MD算法推向成熟。MD5算法经MD2、MD3和MD4算法发展而来，算法复杂程度和安全强度大大提高。但不管是MD2、MD4还是MD5算法，其算法的最终结果均是产生一个128位的消息摘要，这也是MD系列算法的特点。MD5算法执行效率略次于MD4算法，但在安全性方面，MD5算法更胜一筹。随着计算机技术的发展和计算水平的不断提高，MD5算法暴露出来的漏洞也越来越多。MD5算法已不再适合安全要求较高的场合使用。

有关MD5算法的详情请参见RFC 1321（http://www.ietf.org/rfc/rfc1321.txt），其中包含了MD2、MD4和MD5三种算法的C语言版实现。

6.2.2 模型分析

我们以一般Web系统中的用户注册/登录模型为例，分析MD5算法在其中的作用。当然，SHA算法也完全适用这个模型，并且有着更高的安全性。

架构师在设计Web应用中的用户注册/登录模块时，都会考虑这样几个问题：

1. 密码如何传递

（1）以Get方式明文传递，不安全。任何人都可以通过链接分析得到。

（2）以Post方式编码传递，也不安全。编码算法是公开的，任何人都能解码。

2. 密码如何存储

（1）明文存储，不安全。拥有数据库查询权限的人就很可能泄露用户密码。

（2）密文存储，安全。如果对密码做摘要处理，任何人都难以破解。

3. 密码如何校验

（1）明文校验，不安全。与明文存储有同样的隐患。

（2）密文校验，安全。与密文存储有相同的优势。

很明显，三个问题答案选项中，后者更具优势。那么这样一个系统的用户注册与登录是怎样一个流程呢？我们通过图6-2和图6-3进行分析。

对于上述流程，大部分读者朋友都有所体会，作者在这里就不复述了。

用户注册/登录加入消息摘要机制后，有以下几个特点：

1）用户注册时，需要将其密码做摘要处理，以摘要信息存入数据库。

2）用户登录时，需要将其密码做摘要处理，通过数据做查询。

3）用户登录入口处，密码是明文，而数据库中是密文。

4）明文与密文存在对应关系，但仅限于明文到密文的转换，即便是消息摘要算法泄露也不能反推出密码，增强了系统的安全性。

在这样的用户注册/登录模块中，常能见到MD5和SHA算法。为了提高安全性，往往将一些其他的不可变信息，如用户名、Email地址串入原始密码中，使得密码破译的难度加大。其

更多资源请访问稀酷客(www.ckook.com)

中，不可变信息称为“盐”，而这种处理方式常称为“加盐处理”，其实就是对原始信息的一堆混淆。在基于密码的加密算法PBE中，我们将再次看到这种处理方式。

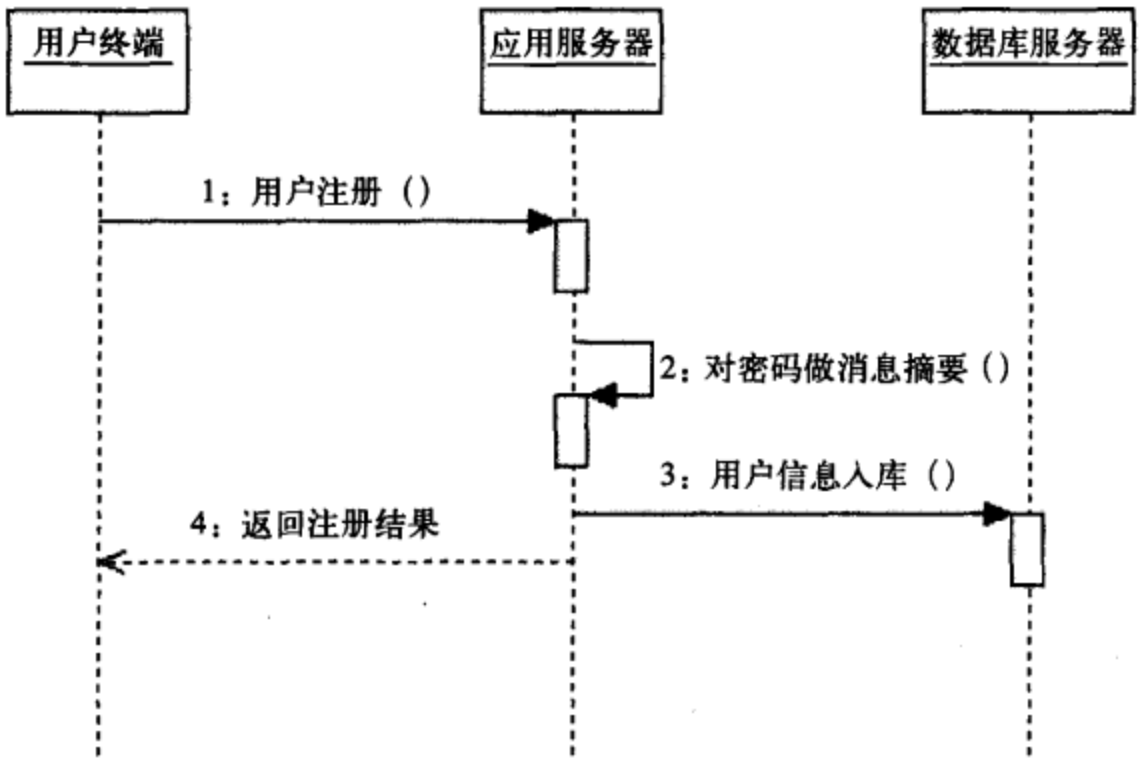


图6-2 用户注册

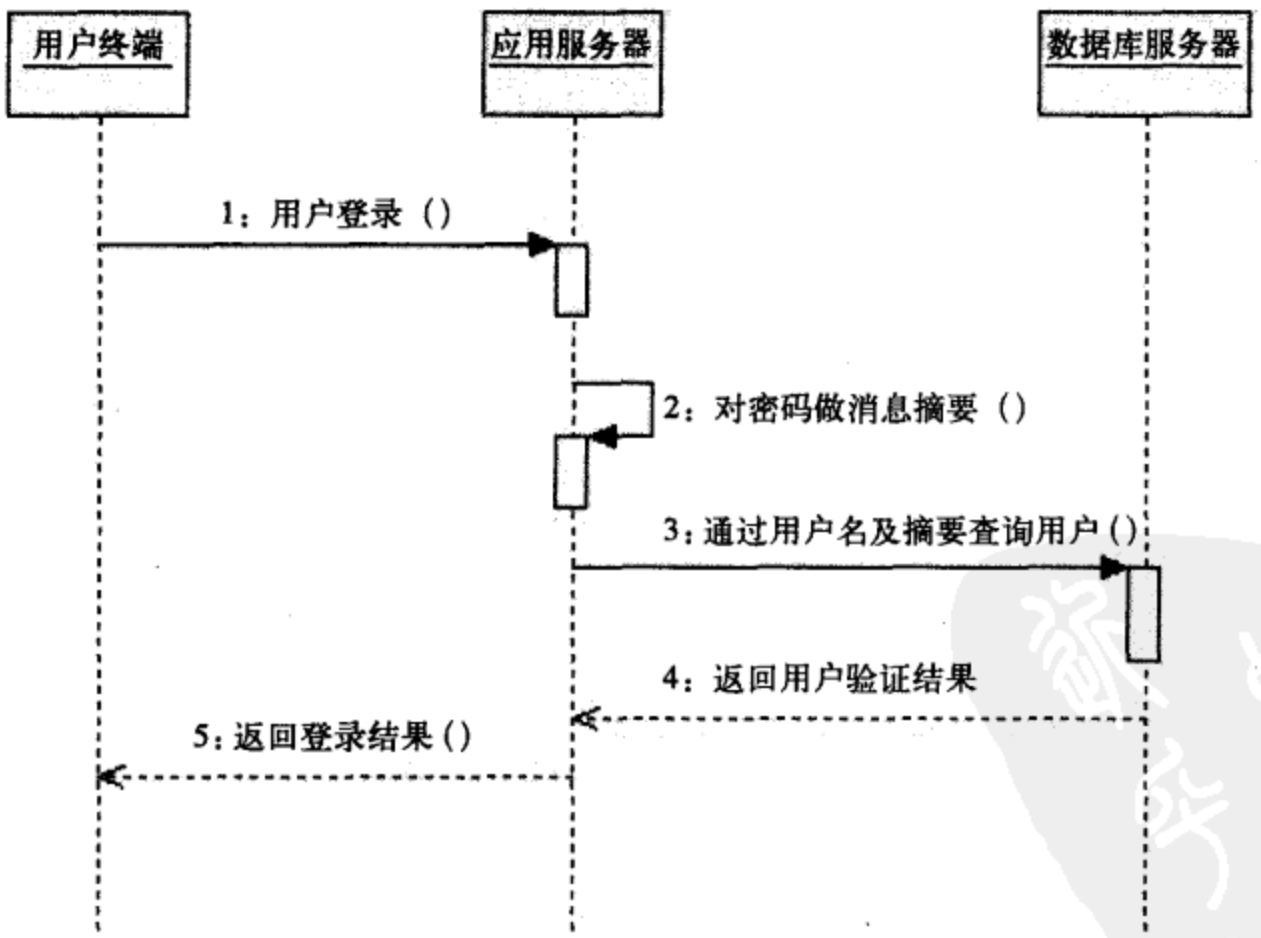


图6-3 用户登录

在一些开源架构中，例如Spring Security（http://static.springsource.org/spring-security/site/），使用了与上述流程相近的设计方案，并提供了MD5和SHA算法支持，以及对加盐处理方式的支持。

更多资源请访问稀酷客(www.ckook.com)

6.2.3 实现

MD系列算法的实现是通过MessageDigest类来完成的，如果需要以流的处理方式完成消息摘要，则需要使用DigestInputStream和DigestOutputStream，有关Java API请读者朋友参照第3章内容。Java 6仅支持MD2和MD5两种算法，通过第三方加密组件包Bouncy Castle（详见第4章），可支持MD4算法。

表6-1 MD系列算法

算法	摘要长度	备注
MD2	128	Java 6实现
MD5		
MD4		Bouncy Castle实现

MD系列算法支持如表6-1所示。

1. Sun

在Java 6中使用MD算法是很简单的。例如，要使用MD5算法对数据做消息摘要，可参考如下代码：

```
// 初始化MessageDigest，并指定MD5算法
MessageDigest md = MessageDigest.getInstance("MD5");
// 摘要处理
byte[] b = md.digest(data);
```

在上述代码中，data[]为待做消息摘要处理的数据，b[]是经过消息摘要处理后的摘要信息，也就是数字指纹。

Java 6支持MD2和MD5算法，如要使用MD2算法只需替换算法名即可，相关实现如代码清单6-1所示。

代码清单6-1 MD2和MD5算法实现

```
import java.security.MessageDigest;
/**
 * MD消息摘要组件
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class MDCoder {
    /**
     * MD2消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeMD2(byte[] data) throws Exception {
            // 初始化MessageDigest
            MessageDigest md = MessageDigest.getInstance("MD2");
            // 执行消息摘要
            return md.digest(data);
    }
```

更多资源请访问稀酷客(www.ckook.com)

```
    /**
     * MD5消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeMD5(byte[] data) throws Exception {
            // 初始化MessageDigest
            MessageDigest md = MessageDigest.getInstance("MD5");
            // 执行消息摘要
            return md.digest(data);
    }
}
```

消息摘要的主要特点就是对同一段数据做多次摘要处理后，其摘要值完全一致。因此，我们通过必要两次消息摘要的结果来判别消息摘要是否一致，见代码清单6-2。

代码清单6-2 MD2和MD5算法实现测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * MD校验
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class MDCoderTest {
    /**
     * 测试MD2
     * @throws Exception
     */
    @Test
    public final void testEncodeMD2() throws Exception {
            String str = "MD2消息摘要";
            // 获得摘要信息
            byte[] data1 = MDCoder.encodeMD2(str.getBytes());
            byte[] data2 = MDCoder.encodeMD2(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
    }
    /**
     * 测试MD5
     * @throws Exception
     */
    @Test
    public final void testEncodeMD5() throws Exception {
            String str = "MD5消息摘要";
```

更多资源请访问稀酷客(www.ckook.com)

```
            // 获得摘要信息
            byte[] data1 = MDCoder.encodeMD5(str.getBytes());
            byte[] data2 = MDCoder.encodeMD5(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
    }
}
```

测试结果自然不用说，一定完全一致！

2. Bouncy Castle

第三方加密组件包Bouncy Castle是对Java 6的友善补充，不仅提供了Base64算法的实现，更是弥补了Sun未能提供MD4算法的空白。

使用Bouncy Castle支持MD4算法比较简单的办法是将Bouncy Castle导入项目中，并在初始化MessageDigest前通过Security类的addProvider()方法加载第三方加密组件包提供者，我们以BouncyCastleProvider为例，提供MD4算法支持。要使用MD4算法对数据做消息摘要，可参考如下代码：

```
import java.security.Security;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
// 省略
// 加入BouncyCastleProvider支持
Security.addProvider(new BouncyCastleProvider());
// 初始化MessageDigest
MessageDigest md = MessageDigest.getInstance("MD4");
// 执行消息摘要
md.digest(data);
```

有关于第三方加密组件包Bouncy Castle的配置，读者朋友可阅读第4章相关内容。下面给出MD4算法实现，如代码清单6-3所示。

代码清单6-3 MD4算法实现

```
import java.security.MessageDigest;
import java.security.Security;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.util.encoders.Hex;
/**
 * MD4消息摘要组件
 * @author梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class MD4Coder {
    /**
     * MD4消息摘要
     * @param data待做摘要处理的数据
     * @return byte[]消息摘要
```

更多资源请访问稀酷客(www.ckook.com)

```
     * @throws Exception
     */
    public static byte[] encodeMD4(byte[] data) throws Exception {
            // 加入BouncyCastleProvider支持
            Security.addProvider(new BouncyCastleProvider());
            // 初始化MessageDigest
            MessageDigest md = MessageDigest.getInstance("MD4");
            // 执行消息摘要
            return md.digest(data);
    }
    /**
     * MD4消息摘要
     * @param data 待做摘要处理的数据
     * @return String 消息摘要
     * @throws Exception
     */
    public static String encodeMD4Hex(byte[] data) throws Exception {
            // 执行消息摘要
            byte[] b = encodeMD4(data);
            // 做十六进制编码处理
            return new String(Hex.encode(b));
    }
}
```

上述代码提供了MD4算法实现（encode MD4 ()方法）的同时，加入了十六进制转换方法实现（encodeMD4Hex()方法）。关于十六进制编码转换相关API请阅读第4章内容。

我们完全可以将Bouncy Castle的相关支持融入其他MD算法实现中，提供完整的MD系列算法，并加入十六进制转换实现。

对于上述方法实现给出对应的测试用例，见代码清单6-4。

代码清单6-4 MD4算法实现测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * MD4校验
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class MD4CoderTest {
    /**
     * 测试MD4
     * @throws Exception
     */
    @Test
    public final void testEncodeMD4() throws Exception {
```

更多资源请访问稀酷客(www.ckook.com)

```
        String str = "MD4消息摘要";
        // 获得摘要信息
        byte[] data1 = MD4Coder.encodeMD4(str.getBytes());
        byte[] data2 = MD4Coder.encodeMD4(str.getBytes());
        // 校验
        assertArrayEquals(data1, data2);
    }
    /**
     * 测试MD4Hex
     * @throws Exception
     */
    @Test
    public final void testEncodeMD4Hex() throws Exception {
        String str = "MD4Hex消息摘要";
        // 获得摘要信息
        String data1 = MD4Coder.encodeMD4Hex(str.getBytes());
        String data2 = MD4Coder.encodeMD4Hex(str.getBytes());
        System.err.println("原文: \t" + str);
        System.err.println("MD4Hex-1: \t" + data1);
        System.err.println("MD4Hex-2: \t" + data2);
        // 校验
        assertEquals(data1, data2);
    }
```

我们来关注一下testEncodeMD4Hex()方法在控制台上的输出结果，如下所示：

```
原文:       MD4Hex消息摘要
MD4Hex-1:  6d46694b818e0bab41eab6783749f963
MD4Hex-2:  6d46694b818e0bab41eab6783749f963
```

我们获得的两个摘要值都是32位的十六进制字符串，并且是一致的。

如果把十六进制转换的实现融入到其他MD算法的实现中，那岂不是相当完美？

3. Commons Codec

对于Commons Codec，想必读者朋友已经不再陌生。它实现了Base64算法，还提供了用于消息摘要的工具类——DigestUtils类（它位于org.apache.commons.codec.digest包中，请读者朋友阅读第4章相关内容。）。DigestUtils类是对Sun提供的MessageDigest类的一次封装，提供了MD5和SHA系列消息摘要算法的实现。

我们通过代码清单6-5来了解如何通过Commons Codec实现MD5算法。

代码清单6-5 MD5算法实现

```
import org.apache.commons.codec.digest.DigestUtils;
/**
 * MD5消息摘要组件
 * @author 梁栋
 * @version 1.0
```

更多资源请访问稀酷客(www.ckook.com)

```
 * @since 1.0
 */
public abstract class MD5Coder {
    /**
     * MD5消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeMD5(String data) throws Exception {
            // 执行消息摘要
            return DigestUtils.md5(data);
    }
    /**
     * MD5消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static String encodeMD5Hex(String data) throws Exception {
            // 执行消息摘要
            return DigestUtils.md5Hex(data);
    }
}
```

虽然Commons Codec只是对Sun提供的MD5算法实现做了一次简单的包装，但着实为我们使用该算法提供了不小的便利。相应的测试用例见代码清单6-6。

代码清单6-6 MD5算法实现测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * MD5校验
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class MD5CoderTest {
    /**
     * 测试MD5
     * @throws Exception
     */
    @Test
    public final void testEncodeMD5() throws Exception {
            String str = "MD5消息摘要";
            // 获得摘要信息
            byte[] data1 = MD5Coder.encodeMD5(str);
```

更多资源请访问稀酷客(www.ckook.com)

```
            byte[] data2 = MD5Coder.encodeMD5(str);
            // 校验
            assertArrayEquals(data1, data2);
    }
    /**
     * 测试MD5Hex
     * @throws Exception
     */
    @Test
    public final void testEncodeMD5Hex() throws Exception {
            String str = "MD5Hex消息摘要";
            // 获得摘要信息
            String data1 = MD5Coder.encodeMD5Hex(str);
            String data2 = MD5Coder.encodeMD5Hex(str);
            System.err.println("原文：\t" + str);
            System.err.println("MD5Hex-1：\t" + data1);
            System.err.println("MD5Hex-2：\t" + data2);
            // 校验
            assertEquals(data1, data2);
    }
}
```

我们来关注一下testEncodeMD5Hex()方法在控制台上的输出结果，如下所示：

```
原文：      MD5Hex消息摘要
MD5Hex-1：  4effa30b56b61156bcd005eb9968cc9b
MD5Hex-2：  4effa30b56b61156bcd005eb9968cc9b
```

对同一个消息做MD5Hex处理后，得到的摘要值都是32位的十六进制字符串，并且是一致的。

4. 三种实现方式的差异

三种实现方式以Sun提供的实现为基础，在算法支持上和方法易用性上提供了更好的扩展与支持。

- Sun

Sun提供的算法实现较为底层，支持MD2和MD5两种算法。但缺少了相应的进制转换实现，不能将其字节数组形式的摘要信息转为十六进制字符串，这多少有点不方便。

- Bouncy Castle

Bouncy Castle是对Sun的友善补充，提供了对MD4算法的支持。支持多种形式的参数，支持十六进制字符串形式的摘要信息。

- Commons Codec

如果仅仅需要实现MD5算法的话，使用Commons Codec完成消息摘要处理是一个不错的选择。它支持多种形式的参数，支持十六进制字符串形式的摘要信息。

综上所述，我们可以根据需要有机地结合两种实现方式，满足系统的需要。如果只要求MD5算法支持，Commons Codec是首选；若要支持MD2或MD4算法，或者要求在此基础上获

更多资源请访问稀酷客(www.ckook.com)

得十六进制编码结果，就让Sun和Bouncy Castle联姻。

6.3 SHA算法家族

SHA算法基于MD4算法基础之上，作为MD算法的继任者，成为了新一代的消息摘要算法的代表。SHA与MD算法不同之处主要在于摘要长度，SHA算法的摘要长度更长，安全性更高。

6.3.1 简述

SHA（Secure Hash Algorithm，安全散列算法）是消息摘要算法的一种，被广泛认可为MD5算法的继任者。它是由美国国家安全局（NSA，National Security Agency）设计，经美国国家标准与技术研究院（NIST，National Institute of Standards and Technology）发布的一系列密码散列函数。SHA算法家族目前共有SHA-1、SHA-224、SHA-256、SHA-384和SHA-512五种算法，通常将后四种算法并称为SHA-2算法。除上述五种算法外，还有发布不久就夭折的SHA-0算法。

SHA算法是在MD4算法的基础上演进而来的，通过SHA算法同样能够获得一个固定长度的摘要信息。与MD系列算法不同的是：若输入的消息不同，则与其相对应的摘要信息的差异概率很高。SHA算法是FIPS所认证的五种安全杂凑算法。

这些算法中的“安全”字眼是基于以下两点（根据官方标准的描述）：

1）由消息摘要反推原输入讯息，从计算理论上来说是很困难的。

2）想要找到两组不同的消息对应到相同的消息摘要，从计算理论上来说也是很困难的。任何对输入消息的变动，都有很高的机率导致其产生的消息摘要迥异。

随着时间的推移，安全算法已不再安全。继山东大学王小云教授顺利破解MD5算法后，SHA-1算法也难逃此劫，终被王小云教授破解。两大著名消息摘要算法被破解，预示着数字签名在理论上可被伪造，B2B和B2C系统将存在安全隐患。

让我们简要回顾一下SHA算法的发展历史：

❑ SHA-0算法

1993年，NIST公布了SHA算法家族的第一个版本，FIPS PUB 180。为避免混淆，现在我们称它为SHA-0算法。但SHA-0算法在公布不久后就被NSA撤回，原因是NSA发现SHA-0算法中含有会降低密码安全性的错误。由此，SHA-0算法还未正式推广就已夭折。

❑ SHA-1算法

1995年，继SHA-0算法夭折后，NIST发布了FIPS PUB 180的修订版本 FIPS PUB 180-1，用于取代FIPS PUB 180，称为SHA-1算法，通常我们也把SHA1算法简称为SHA算法。SHA-1算法在许多安全协定中广为使用，包括TLS/SSL、PGP、SSH、S/MIME 和IPsec，曾被视为是MD5算法的后继者。SHA-0和SHA-1算法可对最大长度为264的字节信息做摘要处理，得到一个160位的摘要信息，其设计原理相似于MD4和MD5算法。如果将得到160位的摘要信息换算成十六进制，可以得到一个40位（每4位二进制数转换为1位十六进制数）的字符串。

更多资源请访问稀酷客(www.ckook.com)

有关SHA-1算法详情请参见RFC 3174（http://www.ietf.org/rfc/rfc3174.txt）。

我们常使用的数字证书就有SHA-1算法的影子，如图6-4所示。图中指纹值正是一个40位的十六进制字符串（51ee11819f269a961671a5bd77cba3f0815103c8）。

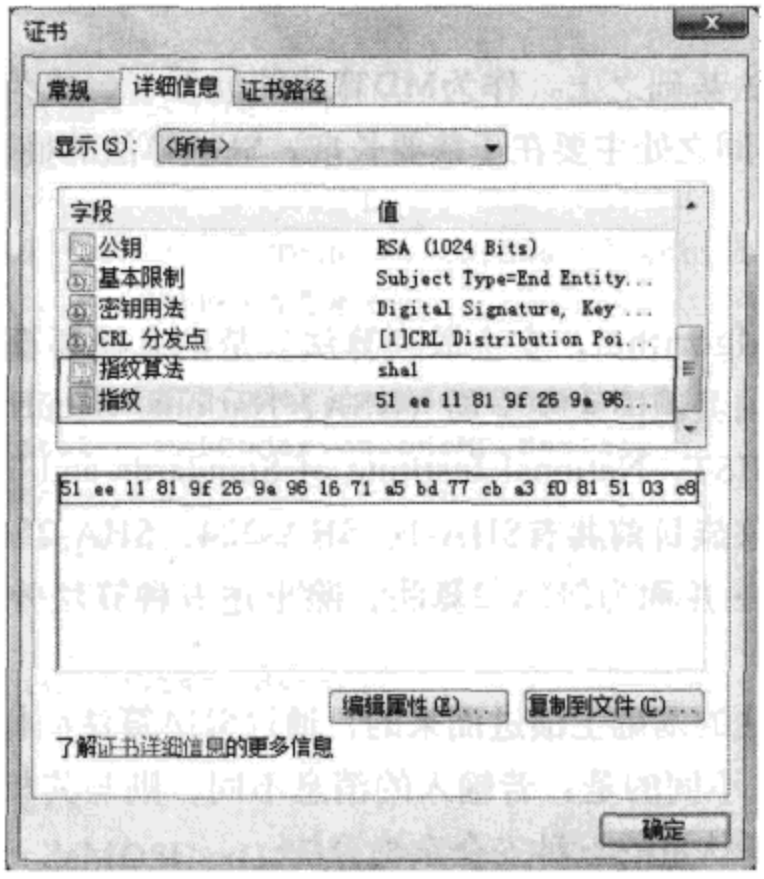


图6-4 数字证书中的SHA-1算法

❑ SHA-2算法

SHA算法家族除了其代表SHA-1算法以外，还有SHA-224、SHA-256、SHA-384和SHA-512四种SHA算法的变体，以其摘要信息字节长度不同而命名，通常将这组算法并称为SHA-2算法。摘要信息字节长度的差异是SHA-2和SHA-1算法的最大差异。

2001年，在FIPS PUB 180-2草稿中包含了SHA-256、SHA-384和SHA-512算法，随即通过了审查和评论，于2002年以官方标准发布。

2004年2月，在FIPS PUB 180-2变更通知中加入了一个额外的变种“SHA-224”，这是为了符合双金钥3DES（三重DES算法）所需的金钥长度而定义的。

6.3.2 模型分析

如果甲乙双方进行明文消息通讯，但要求能够鉴别消息在传输过程中是否被篡改，就可参照如图6-5所示的模型。当然，此模型对MD和SHA算法同样适用。

如图所示，甲乙双方分别作为消息发送者与接收者。我们也可以把甲方看做软件发布厂商，把乙方看做软件使用客户。甲方向乙方发送一则消息，需要经历以下几个步骤：

1）甲方公布消息摘要算法。这个算法可以被其他人获取，包括监听者。

2）甲方对待发送的消息做摘要处理，并获得摘要信息。

3）甲方向乙方发送摘要信息。这个数字指纹可以被其他人获取，包括监听者。

更多资源请访问稀酷客(www.ckook.com)

4）甲方向乙方发送消息。该消息可以是明文，有可能被监听者篡改。

5）乙方获得消息后，使用甲方提供的消息摘要算法对其做摘要处理，获得数字指纹并对比甲方传递过来的数字指纹是否匹配。

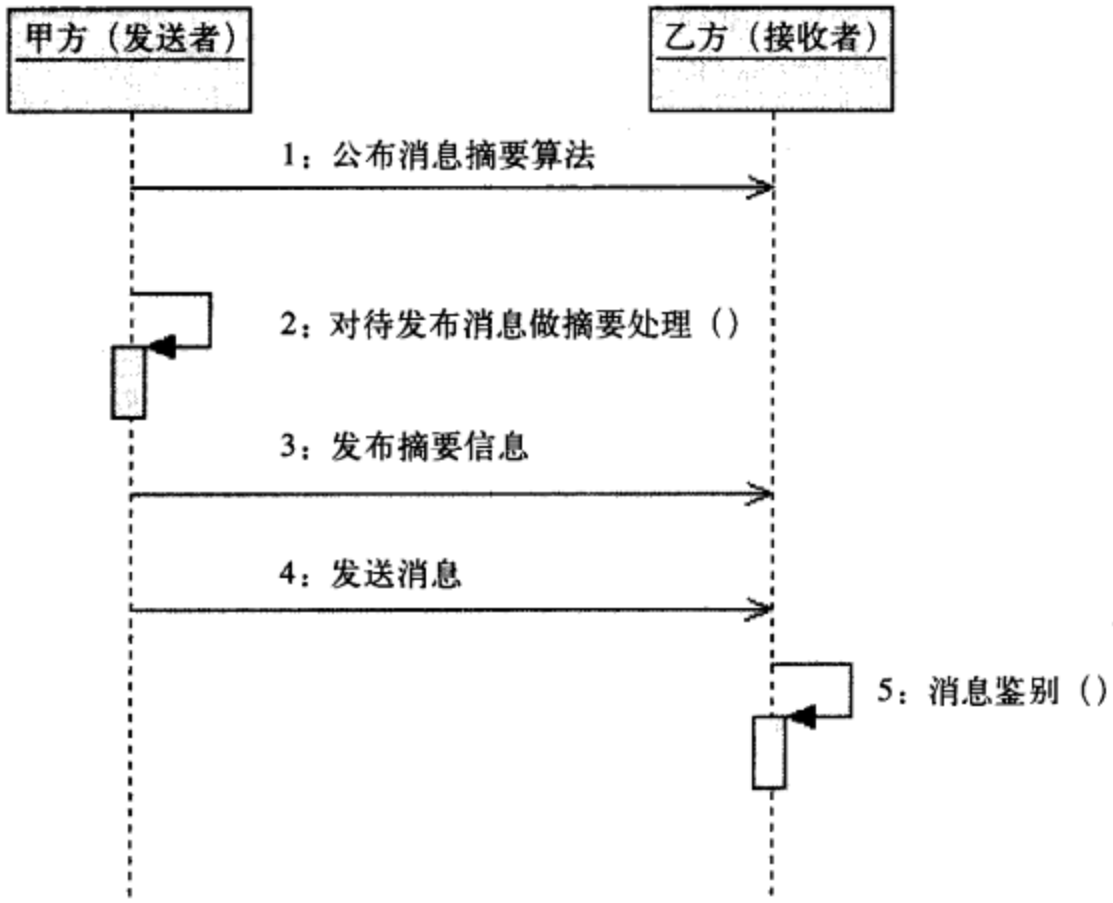


图6-5　基于MD/SHA算法的消息传递

如果两个数字指纹相同则说明消息来源于甲方，否则，甲方传递的消息很可能在传递途中被篡改。

用这个流程图来分析并校验数据的完整性，确定下载到的文件是不是有效的再合适不过了。一般小型的网络交互应用都可参照此模型设计，完成消息的发送与回复。

6.3.3　实现

在Java 6中，MessageDigest类支持MD算法的同时也支持SHA算法，几乎涵盖了我们目前所知的全部SHA系列算法，主要包含SHA-1、SHA-256、SHA-384和SHA-512四种算法。通过第三方加密组件包Bouncy Castle（详见本书附录），可支持SHA-224算法。

SHA系列算法支持如表6-2所示。

表6-2　SHA系列算法

算法	摘要长度	备注
SHA-1	160	Java 6实现
SHA-256	256	
SHA-384	384	
SHA-512	512	
SHA-224	224	Bouncy Castle实现

更多资源请访问稀酷客(www.ckook.com)

1. Sun

在Java 6中，“SHA”是“SHA-1”的简称，两种算法名称等同。如果要使用SHA-1算法对数据做消息摘要，可参考如下代码：

```
// 初始化MessageDigest，并指定SHA算法
MessageDigest md = MessageDigest.getInstance("SHA");
// 摘要处理
byte[] b = md.digest(data);
```

在上述代码中，data[]为待做消息摘要处理的数据，b[]是经过消息摘要处理后的摘要信息。Java 6支持SHA-1、SHA-256、SHA-384和SHA-512四种算法，实现方式如代码清单6-7所示。

代码清单6-7 SHA算法实现1

```
import java.security.MessageDigest;
/**
 * SHA消息摘要组件
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class SHACoder {
    /**
     * SHA-1消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeSHA(byte[] data) throws Exception {
        // 初始化MessageDigest
        MessageDigest md = MessageDigest.getInstance("SHA");
        // 执行消息摘要
        return md.digest(data);
    }
    /**
     * SHA-256消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeSHA256(byte[] data) throws Exception {
        // 初始化MessageDigest
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        // 执行消息摘要
        return md.digest(data);
    }
    /**
     * SHA-384消息摘要
```

更多资源请访问稀酷客(www.ckook.com)

```
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeSHA384(byte[] data) throws Exception {
            // 初始化MessageDigest
            MessageDigest md = MessageDigest.getInstance("SHA-384");
            // 执行消息摘要
            return md.digest(data);
    }
    /**
     * SHA-512消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeSHA512(byte[] data) throws Exception {
            // 初始化MessageDigest
            MessageDigest md = MessageDigest.getInstance("SHA-512");
            // 执行消息摘要
            return md.digest(data);
    }
}
```

对上述代码做测试用例，如代码清单6-8所示。

代码清单6-8 SHA算法实现1测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * SHA校验
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class SHACoderTest {
    /**
     * 测试SHA-1
     * @throws Exception
     */
    @Test
    public final void testEncodeSHA() throws Exception {
            String str = "SHA1消息摘要";
            // 获得摘要信息
            byte[] data1 = SHACoder.encodeSHA(str.getBytes());
            byte[] data2 = SHACoder.encodeSHA(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
```

更多资源请访问稀酷客(www.ckook.com)

```
    }
    /**
     * 测试SHA-256
     * @throws Exception
     */
    @Test
    public final void testEncodeSHA256() throws Exception {
            String str = "SHA256消息摘要";
            // 获得摘要信息
            byte[] data1 = SHACoder.encodeSHA256(str.getBytes());
            byte[] data2 = SHACoder.encodeSHA256(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
    }
    /**
     * 测试SHA-384
     * @throws Exception
     */
    @Test
    public final void testEncodeSHA384() throws Exception {
            String str = "SHA384消息摘要";
            // 获得摘要信息
            byte[] data1 = SHACoder.encodeSHA384(str.getBytes());
            byte[] data2 = SHACoder.encodeSHA384(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
    }
    /**
     * 测试SHA-512
     * @throws Exception
     */
    @Test
    public final void testEncodeSHA512() throws Exception {
            String str = "SHA512消息摘要";
            // 获得摘要信息
            byte[] data1 = SHACoder.encodeSHA512(str.getBytes());
            byte[] data2 = SHACoder.encodeSHA512(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
    }
}
```

作者和读者朋友一定都会这样想：如果能增加十六进制转换的方法，那就更加方便了！

2. Bouncy Castle

Bouncy Castle不仅支持MD4算法，同时也支持SHA-224算法。

比较简单的实现方式如下所示：

```
import java.security.Security;
```

更多资源请访问稀酷客(www.ckook.com)

```
import org.bouncycastle.jce.provider.BouncyCastleProvider;
// 加入BouncyCastleProvider支持
Security.addProvider(new BouncyCastleProvider());
// 初始化MessageDigest
MessageDigest md = MessageDigest.getInstance("SHA-224");
```

有关第三方加密组件包Bouncy Castle，读者可参考第3章和第4章相关内容。

对于SHA算法，我们仅仅需要补充SHA-244算法实现，如代码清单6-9所示。

代码清单6-9 SHA224算法实现

```
import java.security.MessageDigest;
import java.security.Security;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.util.encoders.Hex;
/**
 * SHA-224消息摘要组件
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class SHA224Coder {
    /**
     * SHA-224消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeSHA224(byte[] data) throws Exception {
            // 加入BouncyCastleProvider支持
            Security.addProvider(new BouncyCastleProvider());
            // 初始化MessageDigest
            MessageDigest md = MessageDigest.getInstance("SHA-224");
            // 执行消息摘要
            return md.digest(data);
    }
    /**
     * SHA-224消息摘要
     * @param data 待做摘要处理的数据
     * @return String 十六进制消息摘要
     * @throws Exception
     */
    public static String encodeSHA224Hex(byte[] data) throws Exception {
            // 执行消息摘要
            byte[] b = encodeSHA224(data);
            // 做十六进制编码处理
            return new String(Hex.encode(b));
    }
}
```

更多资源请访问稀酷客(www.ckook.com)

对于上述实现做测试用例也相当简单，如代码清单6-10所示。

代码清单6-10 SHA224算法实现测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * SHA-224校验
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class SHA224CoderTest {
    /**
     * 测试SHA-224
     * @throws Exception
     */
    @Test
    public final void testEncodeSHA224() throws Exception {
            String str = "SHA224消息摘要";
            // 获得摘要信息
            byte[] data1 = SHACoder.encodeSHA224(str.getBytes());
            byte[] data2 = SHACoder.encodeSHA224(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
    }
    /**
     * 测试SHA-224
     * @throws Exception
     */
    @Test
    public final void testEncodeSHA224Hex() throws Exception {
            String str = "SHA224消息摘要";
            // 获得摘要信息
            String data1 = SHA224Coder.encodeSHA224Hex (str.getBytes());
            String data2 = SHA224Coder.encodeSHA224Hex (str.getBytes());
            // 校验
            assertEquals(data1, data2);
    }
}
```

对于其他SHA算法，我们完全可以对其加工，加入十六进制编码转换实现。

我们来关注控制台输出的信息，如下所示：

```
原文:              SHA224Hex消息摘要
SHA224Hex-1:     fd0c016cc823d094aafc4d64665837df8ffa1d1712f58e5c44fd20d2
SHA224Hex-2:     fd0c016cc823d094aafc4d64665837df8ffa1d1712f58e5c44fd20d2
```

更多资源请访问稀酷客(www.ckook.com)

在上述内容中，我们得到了28位的摘要信息，换算成二进制正好是224位。

3. Commons Codec

DigestUtils类除了MD5算法外，还支持多种SHA系列算法，涵盖了Java 6所支持的全部SHA算法。有关DigestUtils类相关SHA算法支持API，读者可阅读第4章内容。

Commons Codec与Sun所提供的SHA算法实现在本质上毫无差别，关键在于Commons Codec提供了更为方便的实现。我们对上述SHA算法实现做了调整，相关实现如代码清单6-11所示。

代码清单6-11 SHA算法实现2

```
import org.apache.commons.codec.digest.DigestUtils;
/**
 * SHA消息摘要组件
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class SHACoder {
    /**
     * SHA消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
 */
public static byte[] encodeSHA(String data) throws Exception {
        // 执行消息摘要
        return DigestUtils.sha(data);
}
/**
 * SHAHex消息摘要
 * @param data 待做摘要处理的数据
 * @return String 消息摘要
 * @throws Exception
 */
public static String encodeSHAHex(String data) throws Exception {
        // 执行消息摘要
        return DigestUtils.shaHex(data);
}
/**
 * SHA256消息摘要
 * @param data 待做摘要处理的数据
 * @return byte[] 消息摘要
 * @throws Exception
 */
public static byte[] encodeSHA256(String data) throws Exception {
        // 执行消息摘要
```

更多资源请访问稀酷客(www.ckook.com)

```
        return DigestUtils.sha256(data);
    }
    /**
     * SHA256Hex消息摘要
     * @param data 待做摘要处理的数据
     * @return String 消息摘要
     * @throws Exception
     */
    public static String encodeSHA256Hex(String data) throws Exception {
        // 执行消息摘要
        return DigestUtils.sha256Hex(data);
    }
    /**
     * SHA384消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeSHA384(String data) throws Exception {
        // 执行消息摘要
        return DigestUtils.sha384(data);
    }
    /**
     * SHA384Hex消息摘要
     * @param data 待做摘要处理的数据
     * @return String 消息摘要
     * @throws Exception
     */
    public static String encodeSHA384Hex(String data) throws Exception {
        // 执行消息摘要
        return DigestUtils.sha384Hex(data);
    }
    /**
     * SHA512Hex消息摘要
     * @param data 待做摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeSHA512(String data) throws Exception {
        // 执行消息摘要
        return DigestUtils.sha512(data);
    }
    /**
     * SHA512Hex消息摘要
     * @param data 待做摘要处理的数据
     * @return String 消息摘要
     * @throws Exception
     */
```

更多资源请访问稀酷客(www.ckook.com)

```
    public static String encodeSHA512Hex(String data) throws Exception {
            // 执行消息摘要
            return DigestUtils.sha512Hex(data);
    }
}
```

对于上述实现做测试用例，如代码清单6-12所示。

代码清单6-12 SHA算法实现2测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * SHAHex校验
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class SHACoderTest {
    /**
     * 测试SHA-1
     * @throws Exception
     */
    @Test
    public final void testEncodeSHA() throws Exception {
        String str = "SHA1消息摘要";
        // 获得摘要信息
        byte[] data1 = SHACoder.encodeSHA(str);
        byte[] data2 = SHACoder.encodeSHA(str);
        // 校验
        assertArrayEquals(data1, data2);
    }
    /**
     * 测试SHA-1Hex
     * @throws Exception
     */
    @Test
    public final void testEncodeSHAHex() throws Exception {
        String str = "SHA-1Hex消息摘要";
        // 获得摘要信息
        String data1 = SHACoder.encodeSHAHex(str);
        String data2 = SHACoder.encodeSHAHex(str);
        System.err.println("原文：\t" + str);
        System.err.println("SHA1Hex-1：\t" + data1);
        System.err.println("SHA1Hex-2：\t" + data2);
        // 校验
        assertEquals(data1, data2);
    }
```

更多资源请访问稀酷客(www.ckook.com)

```
/**
 * 测试SHA-256
 * @throws Exception
 */
@Test
public final void testEncodeSHA256() throws Exception {
        String str = "SHA256消息摘要";
        // 获得摘要信息
        byte[] data1 = SHACoder.encodeSHA256(str);
        byte[] data2 = SHACoder.encodeSHA256(str);
        // 校验
        assertArrayEquals(data1, data2);
}
/**
 * 测试SHA-256Hex
 * @throws Exception
 */
@Test
public final void testEncodeSHA256Hex() throws Exception {
        String str = "SHA256Hex消息摘要";
        // 获得摘要信息
        String data1 = SHACoder.encodeSHA256Hex(str);
        String data2 = SHACoder.encodeSHA256Hex(str);
        System.err.println("原文: \t" + str);
        System.err.println("SHA256Hex-1: \t" + data1);
        System.err.println("SHA256Hex-2: \t" + data2);
        // 校验
        assertEquals(data1, data2);
}
/**
 * 测试SHA-384
 * @throws Exception
 */
@Test
public final void testEncodeSHA384() throws Exception {
        String str = "SHA384消息摘要";
        // 获得摘要信息
        byte[] data1 = SHACoder.encodeSHA384(str);
        byte[] data2 = SHACoder.encodeSHA384(str);
        // 校验
        assertArrayEquals(data1, data2);
}
/**
 * 测试SHA-384Hex
 * @throws Exception
 */
@Test
public final void testEncodeSHA384Hex() throws Exception {
```

更多资源请访问稀酷客(www.ckook.com)

```
        String str = "SHA384Hex消息摘要";
        // 获得摘要信息
        String data1 = SHACoder.encodeSHA384Hex(str);
        String data2 = SHACoder.encodeSHA384Hex(str);
        System.err.println("原文: \t" + str);
        System.err.println("SHA384Hex-1: \t" + data1);
        System.err.println("SHA384Hex-2: \t" + data2);
        // 校验
        assertEquals(data1, data2);
    }
    /**
     * 测试SHA-512
     * @throws Exception
     */
    @Test
    public final void testEncodeSHA512() throws Exception {
        String str = "SHA512消息摘要";
        // 获得摘要信息
        byte[] data1 = SHACoder.encodeSHA512(str);
        byte[] data2 = SHACoder.encodeSHA512(str);
        // 校验
        assertArrayEquals(data1, data2);
    }
    /**
     * 测试SHA-512Hex
     * @throws Exception
     */
    @Test
    public final void testEncodeSHA512Hex() throws Exception {
        String str = "SHA512Hex消息摘要";
        // 获得摘要信息
        String data1 = SHACoder.encodeSHA512Hex(str);
        String data2 = SHACoder.encodeSHA512Hex(str);
        System.err.println("原文: \t" + str);
        System.err.println("SHA512Hex-1: \t" + data1);
        System.err.println("SHA512Hex-2: \t" + data2);
        // 校验
        assertEquals(data1, data2);
    }
}
```

在控制台中，我们可以清晰地观察到4种SHA算法的摘要信息长度有所不同。

❑ SHA-1

```
原文:          SHA-1Hex消息摘要
SHA1Hex-1:    9a4135598d12e61f54d5d790887cf47721cbdd2c
SHA1Hex-2:    9a4135598d12e61f54d5d790887cf47721cbdd2c
```

更多资源请访问稀酷客(www.ckook.com)

SHA-1算法的摘要信息是一个40位的十六进制字符串，换算成二进制正好是160位。

- SHA-256

```
原文:          SHA256Hex消息摘要
SHA256Hex-1: e001852cd945459a14ecd83ec3f9c73b94e02ec12ec8afbf764a98719acce777
SHA256Hex-2: e001852cd945459a14ecd83ec3f9c73b94e02ec12ec8afbf764a98719acce777
```

SHA-256算法的摘要信息是一个64位的十六进制字符串，换算成二进制正好是256位。

- SHA-384和SHA-512

```
原文:          SHA384Hex消息摘要
SHA384Hex-1:
13d87b76ee7a14fcbb7d96c90ed430b8793cdd75afa10700a402ebdce463a29302ce36b19db4f30f
e170399897f191ed
SHA384Hex-2:
13d87b76ee7a14fcbb7d96c90ed430b8793cdd75afa10700a402ebdce463a29302ce36b19db4f30f
e170399897f191ed
原文:          SHA512Hex消息摘要
SHA512Hex-1:
0ecc3742f5ca400c78f84d4bf37a7e34aff371f879d773cab40fe1cd4cb19b6cf2bc598e4bc5a384
808b9b15ef19ff59db4793f0f6b3815bdfd6152208e26756
SHA512Hex-2:
0ecc3742f5ca400c78f84d4bf37a7e34aff371f879d773cab40fe1cd4cb19b6cf2bc598e4bc5a384
808b9b15ef19ff59db4793f0f6b3815bdfd6152208e26756
```

很显然，SHA-384和SHA-512算法的摘要信息没有任何悬念，分别对应384位和512位的二进制数。

消息摘要长度与安全强度成正比。从这一点来说，SHA系列算法比MD系列算法更具优势。MD系列算法仅有128位，而SHA算法则可以从160位扩充到512位，更具安全性。

4. 三种实现方式的差异

三种实现方式以Sun提供的实现为基础，在算法支持上和方法易用性上提供了更好的扩展与支持。

- Sun

由于Sun提供了较为底层的SHA算法实现，如SHA-1、SHA-256、SHA-384和SHA-512四种算法，但缺少了对应的进制转换实现，多少有些遗憾。

- Bouncy Castle

Bouncy Castle是对Sun的友善补充，提供了对SHA-224算法的支持，支持十六进制字符串形式的摘要信息，相当方便。

- Commons Codec

Commons Codec对Sun提供的SHA算法做了包装，支持多种形式的参数，支持十六进制字符串形式的摘要信息，相当方便。

综上所述，若在应用中使用SHA系列算法，且不要求对SHA-224算法提供支持，Commons Codec是更为合适的选择。如果只是需要在Sun原有SHA系列算法支持的基础上加入十六进制编

更多资源请访问稀酷客(www.ckook.com)

码转换，Bouncy Castle和Commons Codec都是不错的选择。

6.4 MAC算法家族

MAC算法结合了MD5和SHA算法的优势，并加入密钥的支持，是一种更为安全的消息摘要算法。

6.4.1 简述

MAC（Message Authentication Code，消息认证码算法）是含有密钥散列函数算法，兼容了MD和SHA算法的特性，并在此基础上加入了密钥。因此，我们也常把MAC称为HMAC (keyed-Hash Message Authentication Code)。

MAC算法主要集合了MD和SHA两大系列消息摘要算法。MD系列算法有HmacMD2、HmacMD4和HmacMD5三种算法；SHA系列算法有HmacSHA1、HmacSHA224、HmacSHA256、HmacSHA384和HmacSHA512五种算法。

经MAC算法得到的摘要值也可以使用十六进制编码表示，其摘要值长度与参与实现的算法摘要值长度相同。例如，HmacSHA1算法得到的摘要长度就是SHA1算法得到的摘要长度，都是160位二进制数，换算成十六进制编码为40位。

有关HMac算法详情请参见RFC 2104（http://www.ietf.org/rfc/rfc2104.txt），其中包含了HmacMD5算法的C语言版实现。

基于SSH（Secure Shell，安全外壳）协议的一些软件，也使用了AES算法。图6-6中展示了SecureCRT软件中如何配置加密算法。

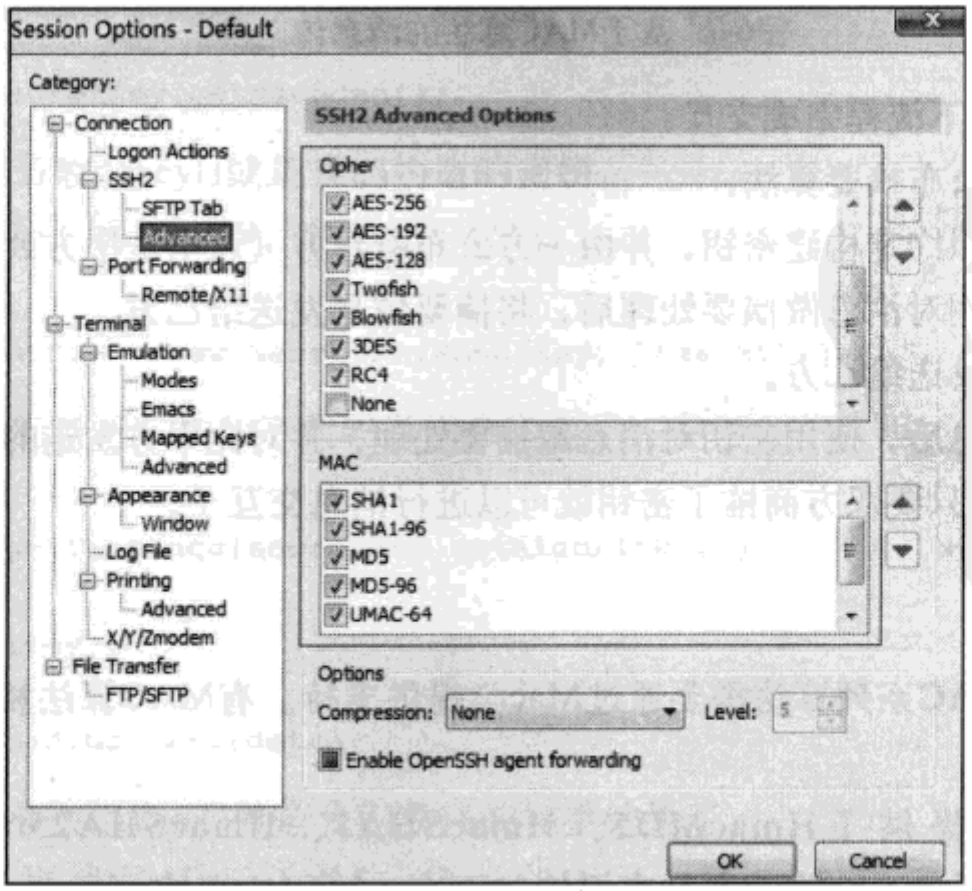


图6-6 SecureCRT配置界面中的加密算法

更多资源请访问稀酷客(www.ckook.com)

6.4.2 模型分析

在本章6.4.2节中，我们分析了基于MD/SHA算法单向消息传递模型中消息摘要算法的作用。如果需要更为安全的消息传递，就需要使用到MAC算法了。基于MAC算法的消息传递模型如图6-7所示。

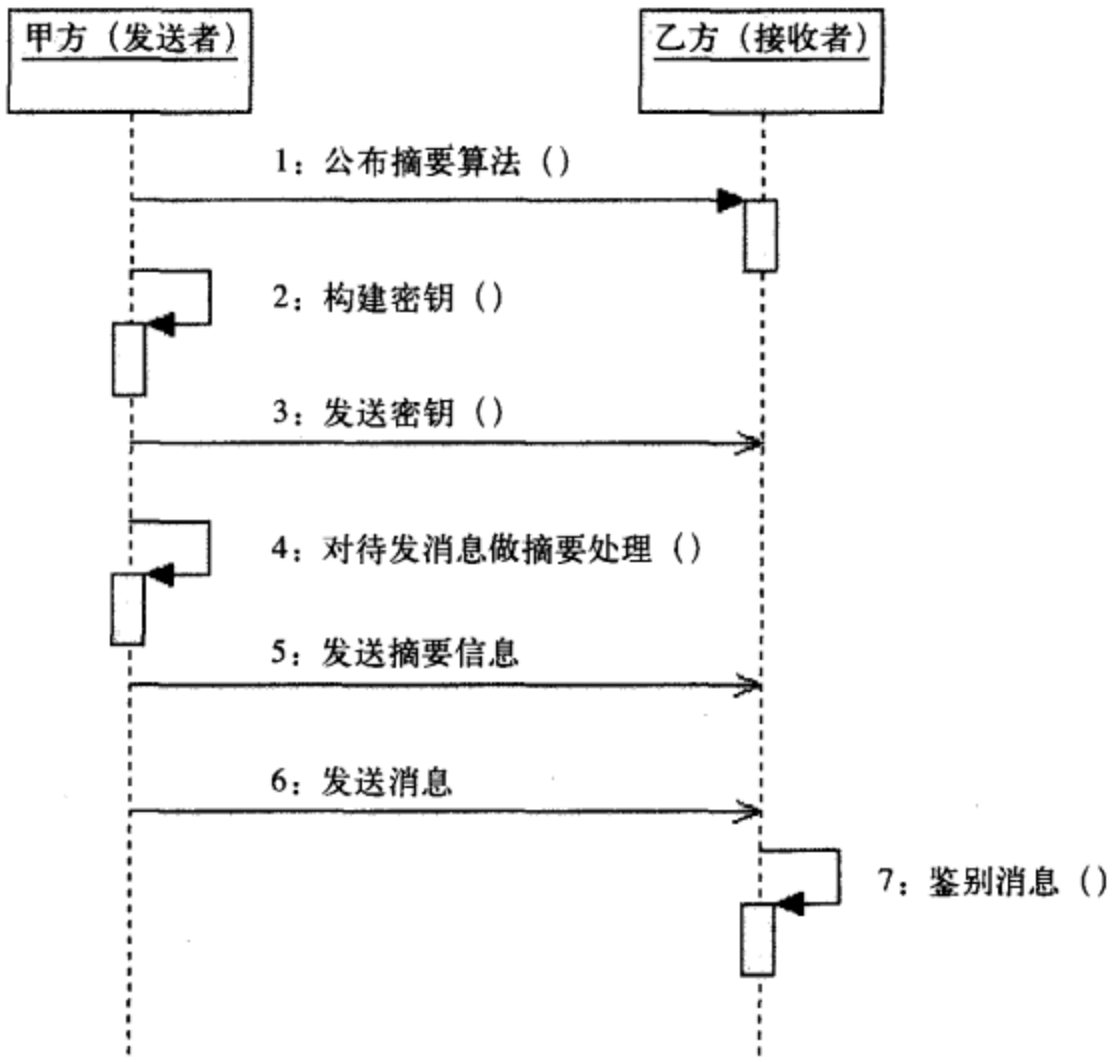


图6-7 基于MAC算法的消息传递模型

甲乙双方可按如下流程完成交互：

1）甲方向乙方公布摘要算法。

2）甲乙双方按照约定构建密钥，并由一方公布给对方（这里是甲方公布密钥给乙方）。

3）甲方使用密钥对消息做摘要处理后，将摘要信息发送给乙方。

4）甲方将消息发送给乙方。

5）乙方收到消息后，使用密钥对消息做摘要处理，并对比甲方发送的摘要信息是否一致。

在上述流程中，只要双方商榷了密钥就可以进行消息交互了。

6.4.3 实现

在Java 6中，MAC系列算法需要通过Mac类提供支持。有MAC算法相关的API请读者朋友参照第3章内容。

Java 6中仅仅提供了HmacMD5、HmacSHA1、HmacSHA256、HmacSHA384和HmacSHA512四种算法，而第三方加密组件包Bouncy Castle补充了HmacMD2、HmacMD4和

更多资源请访问稀酷客(www.ckook.com)

HmacSHA224三种算法支持。有关Bouncy Castle的内容请见附录。

MAC系列算法支持如表6-3所示。

表6-3　MAC系列算法

算法	摘要长度	备注
HmacMD5	128	Java 6实现
HmacSHA1	160	
HmacSHA256	256	
HmacSHA384	384	
HmacSHA512	512	
HmacMD2	128	Bouncy Castle实现
HmacMD4	128	
HmacSHA224	224	

1. Sun

Mac算法是带有密钥的消息摘要算法，所以实现起来要分两步：

1）构建密钥。

2）执行消息摘要。

对应上述步骤，以HmacMD5算法为例，构建密钥代码如下所示：

```
// 初始化KeyGenerator
KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacMD5");
// 产生密钥
SecretKey secretKey = keyGenerator.generateKey();
// 获得密钥
byte[] key = secretKey.getEncoded();
```

上述代码中的字节数组key[]就是我们构造的密钥。

我们需要对其还原，得到密钥，参考如下代码：

```
// 还原密钥
SecretKey secretKey = new SecretKeySpec(key, "HmacMD5");
```

获得密钥后，我们就可以按如下代码做消息摘要了，参考如下代码：

```
// 实例化Mac
Mac mac = Mac.getInstance(secretKey.getAlgorithm());
// 初始化Mac
mac.init(secretKey);
// 执行消息摘要
byte[] data = mac.doFinal(data);
```

上述代码中的字节数组data[]就是我们获得的摘要结果了。

Sun在Java 6中提供了HmacMD5、HmacSHA1、HmacSHA256、HmacSHA384和HmacSHA512四种算法支持，算法实现如代码清单6-13所示。

更多资源请访问稀酷客(www.ckook.com)

代码清单6-13 MAC算法实现1

```
import javax.crypto.KeyGenerator;
import javax.crypto.Mac;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
/**
 * MAC消息摘要组件
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class MACCoder {
    /**
     * 初始化HmacMD5密钥
     * @return byte[] 密钥
     * @throws Exception
     */
    public static byte[] initHmacMD5Key() throws Exception {
            // 初始化KeyGenerator
            KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacMD5");
            // 产生密钥
            SecretKey secretKey = keyGenerator.generateKey();
            // 获得密钥
            return secretKey.getEncoded();
    }
    /**
     * HmacMD5消息摘要
     * @param data 待做摘要处理的数据
     * @param key 密钥
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeHmacMD5(byte[] data, byte[] key)
                    throws Exception {
            // 还原密钥
            SecretKey secretKey = new SecretKeySpec(key, "HmacMD5");
            // 实例化Mac
            Mac mac = Mac.getInstance(secretKey.getAlgorithm());
            // 初始化Mac
            mac.init(secretKey);
            // 执行消息摘要
            return mac.doFinal(data);
    }
    /**
     * 初始化HmacSHA1密钥
     * @return byte[] 密钥
     * @throws Exception
```

更多资源请访问稀酷客(www.ckook.com)

```
     */
    public static byte[] initHmacSHAKey() throws Exception {
            // 初始化KeyGenerator
            KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacSHA1");
            // 产生密钥
            SecretKey secretKey = keyGenerator.generateKey();
            // 获得密钥
            return secretKey.getEncoded();
    }
    /**
     * HmacSHA1消息摘要
     * @param data 待做摘要处理的数据
     * @param key 密钥
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeHmacSHA(byte[] data, byte[] key)
                throws Exception {
            // 还原密钥
            SecretKey secretKey = new SecretKeySpec(key, "HmacSHA1");
            // 实例化Mac
            Mac mac = Mac.getInstance(secretKey.getAlgorithm());
            // 初始化Mac
            mac.init(secretKey);
            // 执行消息摘要
            return mac.doFinal(data);
    }
    /**
     * 初始化HmacSHA256密钥
     * @return byte[] 密钥
     * @throws Exception
     */
    public static byte[] initHmacSHA256Key() throws Exception {
            // 初始化KeyGenerator
            KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacSHA256");
            // 产生密钥
            SecretKey secretKey = keyGenerator.generateKey();
            // 获得密钥
            return secretKey.getEncoded();
    }
    /**
     * HmacSHA256消息摘要
     * @param data 待做摘要处理的数据
     * @param key 密钥
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeHmacSHA256(byte[] data, byte[] key)
```

更多资源请访问稀酷客(www.ckook.com)

```
            throws Exception {
        // 还原密钥
        SecretKey secretKey = new SecretKeySpec(key, "HmacSHA256");
        // 实例化Mac
        Mac mac = Mac.getInstance(secretKey.getAlgorithm());
        // 初始化Mac
        mac.init(secretKey);
        // 执行消息摘要
        return mac.doFinal(data);
}
/**
 * 初始化HmacSHA384密钥
 * @return byte[] 密钥
 * @throws Exception
 */
public static byte[] initHmacSHA384Key() throws Exception {
        // 初始化KeyGenerator
        KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacSHA384");
        // 产生密钥
        SecretKey secretKey = keyGenerator.generateKey();
        // 获得密钥
        return secretKey.getEncoded();
}
/**
 * HmacSHA384消息摘要
 * @param data 待做摘要处理的数据
 * @param key 密钥
 * @return byte[] 消息摘要
 * @throws Exception
 */
public static byte[] encodeHmacSHA384(byte[] data, byte[] key) throws Exception {
        // 还原密钥
        SecretKey secretKey = new SecretKeySpec(key, "HmacSHA384");
        // 实例化Mac
        Mac mac = Mac.getInstance(secretKey.getAlgorithm());
        // 初始化Mac
        mac.init(secretKey);
        // 执行消息摘要
        return mac.doFinal(data);
}
/**
 * 初始化HmacSHA512密钥
 * @return byte[] 密钥
 * @throws Exception
 */
public static byte[] initHmacSHA512Key() throws Exception {
        // 初始化KeyGenerator
```

更多资源请访问稀酷客(www.ckook.com)

```
            KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacSHA512");
            // 产生密钥
            SecretKey secretKey = keyGenerator.generateKey();
            // 获得密钥
            return secretKey.getEncoded();
    }
    /**
     * HmacSHA512消息摘要
     * @param data 待做摘要处理的数据
     * @param key 密钥
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeHmacSHA512(byte[] data, byte[] key) throws Exception {
            // 还原密钥
            SecretKey secretKey = new SecretKeySpec(key, "HmacSHA512");
            // 实例化Mac
            Mac mac = Mac.getInstance(secretKey.getAlgorithm());
            // 初始化Mac
            mac.init(secretKey);
            // 执行消息摘要
            return mac.doFinal(data);
    }
}
```

对于上述代码的测试较为简单，如代码清单6-14所示。

代码清单6-14 MAC算法实现1测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * MAC校验
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class MACCoderTest {
    /**
     * 测试HmacMD5
     * @throws Exception
     */
    @Test
    public final void testEncodeHmacMD5() throws Exception {
            String str = "HmacMD5消息摘要";
            // 初始化密钥
            byte[] key = MACCoder.initHmacMD5Key();
            // 获得摘要信息
            byte[] data1 = MACCoder.encodeHmacMD5(str.getBytes(), key);
```

更多资源请访问稀酷客(www.ckook.com)

```
        byte[] data2 = MACCoder.encodeHmacMD5(str.getBytes(), key);
        // 校验
        assertArrayEquals(data1, data2);
    }
    /**
     * 测试HmacSHA1
     * @throws Exception
     */
    @Test
    public final void testEncodeHmacSHA() throws Exception {
        String str = "HmacSHA1消息摘要";
        // 初始化密钥
        byte[] key = MACCoder.initHmacSHAKey();
        // 获得摘要信息
        byte[] data1 = MACCoder.encodeHmacSHA(str.getBytes(), key);
        byte[] data2 = MACCoder.encodeHmacSHA(str.getBytes(), key);
        // 校验
        assertArrayEquals(data1, data2);
    }
    /**
     * 测试HmacSHA256
     * @throws Exception
     */
    @Test
    public final void testEncodeHmacSHA256() throws Exception {
        String str = "HmacSHA256消息摘要";
        // 初始化密钥
        byte[] key = MACCoder.initHmacSHA256Key();
        // 获得摘要信息
        byte[] data1 = MACCoder.encodeHmacSHA256(str.getBytes(), key);
        byte[] data2 = MACCoder.encodeHmacSHA256(str.getBytes(), key);
        // 校验
        assertArrayEquals(data1, data2);
    }
    /**
     * 测试HmacSHA384
     * @throws Exception
     */
    @Test
    public final void testEncodeHmacSHA384() throws Exception {
        String str = "HmacSHA384消息摘要";
        // 初始化密钥
        byte[] key = MACCoder.initHmacSHA384Key();
        // 获得摘要信息
        byte[] data1 = MACCoder.encodeHmacSHA384(str.getBytes(), key);
        byte[] data2 = MACCoder.encodeHmacSHA384(str.getBytes(), key);
        // 校验
```

更多资源请访问稀酷客(www.ckook.com)

```
            assertArrayEquals(data1, data2);
        }
        /**
         * 测试HmacSHA512
         * @throws Exception
         */
        @Test
        public final void testEncodeHmacSHA512() throws Exception {
            String str = "HmacSHA512消息摘要";
            // 初始化密钥
            byte[] key = MACCoder.initHmacSHA512Key();
            // 获得摘要信息
            byte[] data1 = MACCoder.encodeHmacSHA512(str.getBytes(), key);
            byte[] data2 = MACCoder.encodeHmacSHA512(str.getBytes(), key);
            // 校验
            assertArrayEquals(data1, data2);
        }
    }
```

上述代码实现唯一不足之处在于缺少了十六进制编码转换实现，可以使用Bouncy Castle或Commons Codec的十六进制编码转换实现来做补充。

2. Bouncy Castle

第三方加密组件包Bouncy Castle作为补充，提供了HmacMD2、HmacMD4和HmacSHA224三种算法支持，弥补了Sun在Java 6中未能提供相关算法实现的缺憾。

比较简单的使用方式是将其jar包导入项目中，在做初始化密钥和消息摘要前，执行如下代码：

```
import java.security.Security;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
// 省略
// 加入BouncyCastleProvider支持
Security.addProvider(new BouncyCastleProvider());
```

对于上述算法的实现，如代码清单6-15所示。

代码清单6-15 MAC算法实现2

```
import java.security.Security;
import javax.crypto.KeyGenerator;
import javax.crypto.Mac;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.util.encoders.Hex;
/**
 * MAC消息摘要组件
 * @author 梁栋
 * @version 1.0
```

更多资源请访问稀酷客(www.ckook.com)

```
 * @since 1.0
 */
public abstract class MACCoder {
    /**
     * 初始化HmacMD2密钥
     * @return byte[] 密钥
     * @throws Exception
     */
    public static byte[] initHmacMD2Key() throws Exception {
            // 加入BouncyCastleProvider支持
            Security.addProvider(new BouncyCastleProvider());
            // 初始化KeyGenerator
            KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacMD2");
            // 产生秘密密钥
            SecretKey secretKey = keyGenerator.generateKey();
            // 获得密钥
            return secretKey.getEncoded();
    }
    /**
     * HmacMD2消息摘要
     * @param data 待做消息摘要处理的数据
     * @param key 密钥
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeHmacMD2(byte[] data, byte[] key) throws Exception {
            // 加入BouncyCastleProvider支持
            Security.addProvider(new BouncyCastleProvider());
            // 还原密钥
            SecretKey secretKey = new SecretKeySpec(key, "HmacMD2");
            // 实例化Mac
            Mac mac = Mac.getInstance(secretKey.getAlgorithm());
            // 初始化Mac
            mac.init(secretKey);
            // 执行消息摘要
            return mac.doFinal(data);
    }
    /**
     * HmacMD2Hex消息摘要
     * @param data 待做消息摘要处理的数据
     * @param String 密钥
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static String encodeHmacMD2Hex(byte[] data, byte[] key) throws Exception {
            // 执行消息摘要
            byte[] b = encodeHmacMD2(data, key);
            // 做十六进制转换
```

更多资源请访问稀酷客(www.ckook.com)

```
        return new String(Hex.encode(b));
    }
    /**
     * 初始化HmacMD4密钥
     * @return byte[] 密钥
     * @throws Exception
     */
    public static byte[] initHmacMD4Key() throws Exception {
        // 加入BouncyCastleProvider支持
        Security.addProvider(new BouncyCastleProvider());
        // 初始化KeyGenerator
        KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacMD4");
        // 产生秘密密钥
        SecretKey secretKey = keyGenerator.generateKey();
        // 获得密钥
        return secretKey.getEncoded();
    }
    /**
     * HmacMD4消息摘要
     * @param data 待做消息摘要处理的数据
     * @param key 密钥
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeHmacMD4(byte[] data, byte[] key) throws Exception {
        // 加入BouncyCastleProvider支持
        Security.addProvider(new BouncyCastleProvider());
        // 还原密钥
        SecretKey secretKey = new SecretKeySpec(key, "HmacMD4");
        // 实例化Mac
        Mac mac = Mac.getInstance(secretKey.getAlgorithm());
        // 初始化Mac
        mac.init(secretKey);
        // 执行消息摘要
        return mac.doFinal(data);
    }
    /**
     * HmacMD4Hex消息摘要
     * @param data 待做消息摘要处理的数据
     * @param key 密钥
     * @return String 消息摘要
     * @throws Exception
     */
    public static String encodeHmacMD4Hex(byte[] data, byte[] key) throws Exception {
        // 执行消息摘要
        byte[] b = encodeHmacMD4(data, key);
```

更多资源请访问稀酷客(www.ckook.com)

```
        // 做十六进制转换
        return new String(Hex.encode(b));
    }
    /**
     * 初始化HmacSHA224密钥
     * @return byte[] 密钥
     * @throws Exception
     */
    public static byte[] initHmacSHA224Key() throws Exception {
        // 加入BouncyCastleProvider支持
        Security.addProvider(new BouncyCastleProvider());
        // 初始化KeyGenerator
        KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacSHA224");
        // 产生秘密密钥
        SecretKey secretKey = keyGenerator.generateKey();
        // 获得密钥
        return secretKey.getEncoded();
    }
    /**
     * HmacSHA224消息摘要
     * @param data 待做消息摘要处理的数据
     * @param key 密钥
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeHmacSHA224(byte[] data, byte[] key) throws Exception {
        // 加入BouncyCastleProvider支持
        Security.addProvider(new BouncyCastleProvider());
        // 还原密钥
        SecretKey secretKey = new SecretKeySpec(key, "HmacSHA224");
        // 实例化Mac
        Mac mac = Mac.getInstance(secretKey.getAlgorithm());
        // 初始化Mac
        mac.init(secretKey);
        // 执行消息摘要
        return mac.doFinal(data);
    }
    /**
     * HmacSHA224Hex消息摘要
     * @param data 待做消息摘要处理的数据
     * @param key 密钥
     * @return String 消息摘要
     * @throws Exception
     */
    public static String encodeHmacSHA224Hex(byte[] data, byte[] key) throws Exception {
        // 执行消息摘要
```

更多资源请访问稀酷客(www.ckook.com)

```
            byte[] b = encodeHmacSHA224(data, key);
            // 做十六进制转换
            return new String(Hex.encode(b));
    }
}
```

对于上述实现做相关测试，如代码清单6-16所示。

代码清单6-16 MAC算法实现2测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * MAC校验
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class MACCoderTest {
    /**
     * 测试HmacMD2
     * @throws Exception
     */
    @Test
    public final void testEncodeHmacMD2() throws Exception {
            String str = "HmacMD2消息摘要";
            // 初始化密钥
            byte[] key = MACCoder.initHmacMD2Key();
            // 获得摘要信息
            byte[] data1 = MACCoder.encodeHmacMD2(str.getBytes(), key);
            byte[] data2 = MACCoder.encodeHmacMD2(str.getBytes(), key);
            // 校验
            assertArrayEquals(data1, data2);
    }
    /**
     * 测试HmacMD4
     * @throws Exception
     */
    @Test
    public final void testEncodeHmacMD4() throws Exception {
            String str = "HmacMD4消息摘要";
            // 初始化密钥
            byte[] key = MACCoder.initHmacMD4Key();
            // 获得摘要信息
            byte[] data1 = MACCoder.encodeHmacMD4(str.getBytes(), key);
            byte[] data2 = MACCoder.encodeHmacMD4(str.getBytes(), key);
            // 校验
            assertArrayEquals(data1, data2);
```

更多资源请访问稀酷客(www.ckook.com)

```
    }
    /**
     * 测试HmacSHA224
     * @throws Exception
     */
    @Test
    public final void testEncodeHmacSHA224() throws Exception {
            String str = "HmacSHA224消息摘要";
            // 初始化密钥
            byte[] key = MACCoder.initHmacSHA224Key();
            // 获得摘要信息
            byte[] data1 = MACCoder.encodeHmacSHA224(str.getBytes(), key);
            byte[] data2 = MACCoder.encodeHmacSHA224(str.getBytes(), key);
            // 校验
            assertArrayEquals(data1, data2);
    }
}
```

观察控制台输出的信息，如下所示：

```
原文:                HmacMD2Hex消息摘要
HmacMD2Hex-1:        bf5fa06c2c4855825a23ee08206c892f
HmacMD2Hex-2:        bf5fa06c2c4855825a23ee08206c892f
原文:                HmacMD4Hex消息摘要
HmacMD4Hex-1:        c6ac3ec24690011bdfad9ce2e7aed1e7
HmacMD4Hex-2:        c6ac3ec24690011bdfad9ce2e7aed1e7
原文:                HmacSHA224Hex消息摘要
HmacSHA224Hex-1:     54786b4722f72a2599ebb1bba3732bca9f0353392e51729bfca91c78
HmacSHA224Hex-2:     54786b4722f72a2599ebb1bba3732bca9f0353392e51729bfca91c78
```

我们可以清楚地看到，经HmacMD2Hex 和HmacMD4Hex处理后得到的字符串是一个32位的十六进制字符串，换算成二进制正好是128位，也就是MD系列算法做消息摘要处理后得到的摘要值长度。同理，HmacSHA224Hex处理得到的消息摘要值与SHA-224算法做消息摘要处理后得到的摘要值长度相同。

3. 两种实现方式的差异

两种实现方式以Sun提供的实现为基础，在算法支持上提供了更好的扩展。

- Sun

提供了基本的算法支持，如HmacMD5、HmacSHA1、HmacSHA256、HmacSHA384和Hmac512五种算法支持。

- Bouncy Castle

Bouncy Castle在Sun的基础上添加了对HmacMD2、HmacMD4和HmacSHA224三种算法的支持，同时支持十六进制编码。

综上所述，根据需求恰当使用上述实现是很有必要的。如果需要HmacMD2、HmacMD4或HmacSHA224算法支持，可以使用Bouncy Castle做相关实现。如果还需要对摘要结果做十六进

更多资源请访问稀酷客(www.ckook.com)

制编码，则使用Bouncy Castle更为恰当。

6.5 其他消息摘要算法

除了MD、SHA和MAC这三大主流消息摘要算法外，还有许多我们不了解的消息摘要算法，包括RipeMD系列（包含RipeMD128、RipeMD160、RipeMD256和RipeMD320）、Tiger、Whirlpool和GOST3411算法。

RipeMD系列算法与MAC系列算法相结合，又产生了HmacRipeMD128和HmacRipeMD160两种算法。

作为了解，我们在这里做简要介绍。也许有一天，我们会需要这些非主流消息摘要算法。

6.5.1 简述

作者未能获得上述4种算法的相关定义，在此简要介绍。

❑ RipeMD

RipeMD（RACE Integrity Primitives Evaluation Message Digest），是由Hans Dobbertin等3人在对MD4，MD5缺陷分析的基础上，于1996年提出。目前，RipeMD算法共有4个标准，主要是对摘要值长度的区分，类似于SHA系列算法，包含RipeMD128、RipeMD160、RipeMD256和RipeMD320共4种算法。HmacRipeMD128和HmacRipeMD160算法是RipeMD与MAC算法融合的产物。

❑ Tiger

Tiger由Ross于1995年提出。Tiger号称是最快的Hash算法，专门为64位机器做了优化，其消息摘要长度为192位。

❑ Whirlpool

Whirlpool已被列入ISO标准，由于它使用了与AES加密标准相同的转化技术，极大地提高了安全性，被称为最安全的摘要算法。Whirlpool在历史上共有3个版本，目前最新的版本是2003年颁布的，通常将其称为Whirlpool 3.0，其消息摘要长度为512位。

❑ GOST3411

对于GOST3411算法，作者未能获得更多的相关信息，只能得知该算法得到的摘要信息长度为256位。

6.5.2 实现

Java 6内并没有提供上述算法的实现，这里仅仅是Bouncy Castle的专场。为避免内容冗长，作者在这里以RipeMD系列算法和HmacRipeMD系列算法为例，做简要介绍。

RipeMD系列算法和HmacRipeMD系列算法如表6-4所示。

更多资源请访问稀酷客(www.ckook.com)

表6-4 RipeMD系列算法和HmacRipeMD系列算法

算法	摘要长度	备注
RipeMD128	128	Bouncy Castle实现
RipeMD160	160	
RipeMD256	256	
RipeMD320	320	
HmacRipeMD128	128	
HmacRipeMD160	160	

1. RipeMD系列算法

RipeMD算法的实现具有代表性，Tiger、Whirlpool和GOST3411算法实现与其别无二致。

目前，Bouncy Castle提供了RipeMD128、RipeMD160、RipeMD256和RipeMD320共4种算法实现，代码实现如代码清单6-17所示。

代码清单6-17 RipeMD消息摘要算法实现

```
import java.security.MessageDigest;
import java.security.Security;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.util.encoders.Hex;
/**
 * RipeMD系列消息摘要组件
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class RipeMDCoder {
    /**
     * RipeMD128消息摘要
     * @param data 待做消息摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeRipeMD128(byte[] data) throws Exception {
            // 加入BouncyCastleProvider支持
            Security.addProvider(new BouncyCastleProvider());
            // 初始化MessageDigest
            MessageDigest md = MessageDigest.getInstance("RipeMD128");
            // 执行消息摘要
            return md.digest(data);
    }
    /**
     * RipeMD128Hex消息摘要
     * @param data 待做消息摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
```

更多资源请访问稀酷客(www.ckook.com)

```
     */
    public static String encodeRipeMD128Hex(byte[] data) throws Exception {
            // 执行消息摘要
            byte[] b = encodeRipeMD128(data);
            // 做十六进制编码处理
            return new String(Hex.encode(b));
    }
    /**
     * RipeMD160消息摘要
     * @param data 待做消息摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeRipeMD160(byte[] data) throws Exception {
            // 加入BouncyCastleProvider支持
            Security.addProvider(new BouncyCastleProvider());
            // 初始化MessageDigest
            MessageDigest md = MessageDigest.getInstance("RipeMD160");
            // 执行消息摘要
            return md.digest(data);
    }
    /**
     * RipeMD160Hex消息摘要
     * @param data 待做消息摘要处理的数据
     * @return String 消息摘要
     * @throws Exception
     */
    public static String encodeRipeMD160Hex(byte[] data) throws Exception {
            // 执行消息摘要
            byte[] b = encodeRipeMD160(data);
            // 做十六进制编码处理
            return new String(Hex.encode(b));
    }
    /**
     * RipeMD256消息摘要
     * @param data 待做消息摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeRipeMD256(byte[] data) throws Exception {
            // 加入BouncyCastleProvider支持
            Security.addProvider(new BouncyCastleProvider());
            // 初始化MessageDigest
            MessageDigest md = MessageDigest.getInstance("RipeMD256");
            // 执行消息摘要
            return md.digest(data);
    }
    /**
     * RipeMD256Hex消息摘要
```

更多资源请访问稀酷客(www.ckook.com)

```
     * @param data 待做消息摘要处理的数据
     * @return String 消息摘要
     * @throws Exception
     */
    public static String encodeRipeMD256Hex(byte[] data) throws Exception {
            // 执行消息摘要
            byte[] b = encodeRipeMD256(data);
            // 做十六进制编码处理
            return new String(Hex.encode(b));
    }
    /**
     * RipeMD320消息摘要
     * @param data 待做消息摘要处理的数据
     * @return byte[] 消息摘要
     * @throws Exception
     */
    public static byte[] encodeRipeMD320(byte[] data) throws Exception {
            // 加入BouncyCastleProvider支持
            Security.addProvider(new BouncyCastleProvider());
            // 初始化MessageDigest
            MessageDigest md = MessageDigest.getInstance("RipeMD320");
            // 执行消息摘要
            return md.digest(data);
    }
    /**
     * RipeMD320Hex消息摘要
     * @param data 待做消息摘要处理的数据
     * @return String 消息摘要
     * @throws Exception
     */
    public static String encodeRipeMD320Hex(byte[] data) throws Exception {
            // 执行消息摘要
            byte[] b = encodeRipeMD320(data);
            // 做十六进制编码处理
            return new String(Hex.encode(b));
    }
}
```

上述代码对应的测试用例如代码清单6-18所示。

代码清单6-18 RipeMD消息摘要算法实现测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * RipeMD校验
 * @author 梁栋
 * @version 1.0
 * @since 1.0
```

更多资源请访问稀酷客(www.ckook.com)

```
 */
public class RipeMDCoderTest {
    /**
     * 测试RipeMD128
     * @throws Exception
     */
    @Test
    public final void testEncodeRipeMD128() throws Exception {
            String str = "RipeMD128消息摘要";
            // 获得摘要信息
            byte[] data1 = RipeMDCoder.encodeRipeMD128(str.getBytes());
            byte[] data2 = RipeMDCoder.encodeRipeMD128(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
    }
    /**
     * 测试RipeMD128Hex
     * @throws Exception
     */
    @Test
    public final void testEncodeRipeMD128Hex() throws Exception {
            String str = "RipeMD128Hex消息摘要";
            // 获得摘要信息
            String data1 = RipeMDCoder.encodeRipeMD128Hex(str.getBytes());
            String data2 = RipeMDCoder.encodeRipeMD128Hex(str.getBytes());
            System.err.println("原文:\t" + str);
            System.err.println("RipeMD128Hex-1:\t" + data1);
            System.err.println("RipeMD128Hex-2:\t" + data2);
            // 校验
            assertEquals(data1, data2);
    }
    /**
     * 测试RipeMD160
     * @throws Exception
     */
    @Test
    public final void testEncodeRipeMD160() throws Exception {
            String str = "RipeMD160消息摘要";
            // 获得摘要信息
            byte[] data1 = RipeMDCoder.encodeRipeMD160(str.getBytes());
            byte[] data2 = RipeMDCoder.encodeRipeMD160(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
    }
    /**
     * 测试RipeMD160Hex
     * @throws Exception
     */
    @Test
```

更多资源请访问稀酷客(www.ckook.com)

```
    public final void testEncodeRipeMD160Hex() throws Exception {
            String str = "RipeMD160Hex消息摘要";
            // 获得摘要信息
            String data1 = RipeMDCoder.encodeRipeMD160Hex(str.getBytes());
            String data2 = RipeMDCoder.encodeRipeMD160Hex(str.getBytes());
            System.err.println("原文: \t" + str);
            System.err.println("RipeMD160Hex-1: \t" + data1);
            System.err.println("RipeMD160Hex-2: \t" + data2);
            // 校验
            assertEquals(data1, data2);
    }
    /**
     * 测试RipeMD256
     * @throws Exception
     */
    @Test
    public final void testEncodeRipeMD256() throws Exception {
            String str = "RipeMD256消息摘要";
            // 获得摘要信息
            byte[] data1 = RipeMDCoder.encodeRipeMD256(str.getBytes());
            byte[] data2 = RipeMDCoder.encodeRipeMD256(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
    }
    /**
     * 测试RipeMD256Hex
     * @throws Exception
     */
    @Test
    public final void testEncodeRipeMD256Hex() throws Exception {
            String str = "RipeMD256Hex消息摘要";
            // 获得摘要信息
            String data1 = RipeMDCoder.encodeRipeMD256Hex(str.getBytes());
            String data2 = RipeMDCoder.encodeRipeMD256Hex(str.getBytes());
            System.err.println("原文: \t" + str);
            System.err.println("RipeMD256Hex-1: \t" + data1);
            System.err.println("RipeMD256Hex-2: \t" + data2);
            // 校验
            assertEquals(data1, data2);
    }
    /**
     * 测试RipeMD320
     * @throws Exception
     */
    @Test
    public final void testEncodeRipeMD320() throws Exception {
            String str = "RipeMD320消息摘要";
            // 获得摘要信息
            byte[] data1 = RipeMDCoder.encodeRipeMD320(str.getBytes());
```

更多资源请访问稀酷客(www.ckook.com)

```
            byte[] data2 = RipeMDCoder.encodeRipeMD320(str.getBytes());
            // 校验
            assertArrayEquals(data1, data2);
    }
    /**
     * 测试RipeMD320Hex
     * @throws Exception
     */
    @Test
    public final void testEncodeRipeMD320Hex() throws Exception {
            String str = "RipeMD320Hex消息摘要";
            // 获得摘要信息
            String data1 = RipeMDCoder.encodeRipeMD320Hex(str.getBytes());
            String data2 = RipeMDCoder.encodeRipeMD320Hex(str.getBytes());
            System.err.println("原文: \t" + str);
            System.err.println("RipeMD320Hex-1: \t" + data1);
            System.err.println("RipeMD320Hex-2: \t" + data2);
            // 校验
            assertEquals(data1, data2);
    }
}
```

我们一起来观察控制台输出的信息。

以下是RipeMD128处理后的十六进制摘要结果：

```
原文:                   RipeMD128Hex消息摘要
RipeMD128Hex-1:     3a7d75b69093be33f7f5442b97e09d69
RipeMD128Hex-2:     3a7d75b69093be33f7f5442b97e09d69
```

我们得到了一个32位的十六进制字符串，换算成二进制正好是128位。

对于RipeMD160、RipeMD256和RipeMD320算法的摘要值的十六进制自然是40位、64位和80位。

以下是RipeMD160算法的控制台输出信息：

```
原文:                   RipeMD160Hex消息摘要
RipeMD160Hex-1:   5baec3e74e72e719fc702bc0057d6bb80c037ee3
RipeMD160Hex-2:   5baec3e74e72e719fc702bc0057d6bb80c037ee3
```

以下是RipeMD256算法的控制台输出信息：

```
原文:                   RipeMD256Hex消息摘要
RipeMD256Hex-1:  466776542909d4e57215deaaad7f7a50351c6250a0fd3bdddf20cfa28a5f6678
RipeMD256Hex-2:  466776542909d4e57215deaaad7f7a50351c6250a0fd3bdddf20cfa28a5f6678
```

以下是RipeMD320算法的控制台输出信息：

```
原文:               RipeMD320Hex消息摘要
RipeMD320Hex-1: e8e9c116bf1b8762199c1b411f6c4d0bd3ac34fc448736e9ccbb91144f8ec279e55da92c7d3c847c
RipeMD320Hex-2: e8e9c116bf1b8762199c1b411f6c4d0bd3ac34fc448736e9ccbb91144f8ec279e55da92c7d3c847c
```

对于Tiger、Whirlpool和GOST3411算法的实现与测试用例，与RipeMD无任何差别，只需

更多资源请访问稀酷客(www.ckook.com)

替换算法名称即可，作者在这里就不再复述了。

2. HmacRipeMD系列算法

HmacRipeMD算法是RipeMD与MAC两种算法的融合，实现起来较为简单。目前，Bouncy Castle提供了HmacRipeMD128和HmacRipeMD160两种算法实现，代码实现如代码清单6-19所示。

代码清单6-19 HmacRipeMD系列算法实现

```
import java.security.Security;
import javax.crypto.KeyGenerator;
import javax.crypto.Mac;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.util.encoders.Hex;
/**
 * HmacRipeMD系列消息摘要组件
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class HmacRipeMDCoder {
/**
 * 初始化HmacRipeMD128密钥
 * @return byte[] 密钥
 * @throws Exception
 */
public static byte[] initHmacRipeMD128Key() throws Exception {
        // 加入BouncyCastleProvider支持
        Security.addProvider(new BouncyCastleProvider());
        // 初始化KeyGenerator
        KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacRipeMD128");
        // 产生秘密密钥
        SecretKey secretKey = keyGenerator.generateKey();
        // 获得密钥
        return secretKey.getEncoded();
}
/**
 * HmacRipeMD128消息摘要
 * @param data 待做消息摘要处理的数据
 * @param key 密钥
 * @return byte[] 消息摘要
 * @throws Exception
 */
public static byte[] encodeHmacRipeMD128(byte[] data, byte[] key)
            throws Exception {
        // 加入BouncyCastleProvider支持
```

更多资源请访问稀酷客(www.ckook.com)

```
        Security.addProvider(new BouncyCastleProvider());
        // 还原密钥
        SecretKey secretKey = new SecretKeySpec(key, "HmacRipeMD128");
        // 实例化Mac
        Mac mac = Mac.getInstance(secretKey.getAlgorithm());
        // 初始化Mac
        mac.init(secretKey);
        // 执行消息摘要
        return mac.doFinal(data);
}
/**
 * HmacRipeMD128Hex消息摘要
 * @param data 待做消息摘要处理的数据
 * @param key 密钥
 * @return String 消息摘要
 * @throws Exception
 */
public static String encodeHmacRipeMD128Hex(byte[] data, byte[] key)
            throws Exception {
        // 执行消息摘要
        byte[] b = encodeHmacRipeMD128(data, key);
        // 做十六进制转换
        return new String(Hex.encode(b));
}
/**
 * 初始化HmacRipeMD160密钥
 * @return byte[] 密钥
 * @throws Exception
 */
public static byte[] initHmacRipeMD160Key() throws Exception {
        // 加入BouncyCastleProvider支持
        Security.addProvider(new BouncyCastleProvider());
        // 初始化KeyGenerator
        KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacRipeMD160");
        // 产生秘密密钥
        SecretKey secretKey = keyGenerator.generateKey();
        // 获得密钥
        return secretKey.getEncoded();
}
/**
 * HmacRipeMD160消息摘要
 * @param data 待做消息摘要处理的数据
 * @param key 密钥
 * @return byte[] 消息摘要
 * @throws Exception
 */
public static byte[] encodeHmacRipeMD160(byte[] data, byte[] key)
            throws Exception {
```

更多资源请访问稀酷客(www.ckook.com)

```
        // 加入BouncyCastleProvider支持
        Security.addProvider(new BouncyCastleProvider());
        // 还原密钥
        SecretKey secretKey = new SecretKeySpec(key, "HmacRipeMD160");
        // 实例化Mac
        Mac mac = Mac.getInstance(secretKey.getAlgorithm());
        // 初始化Mac
        mac.init(secretKey);
        // 执行消息摘要
        return mac.doFinal(data);
    }
    /**
     * HmacRipeMD160Hex消息摘要
     * @param data 待做消息摘要处理的数据
     * @param key 密钥
     * @return String 消息摘要
     * @throws Exception
     */
    public static String encodeHmacRipeMD160Hex(byte[] data, byte[] key)
            throws Exception {
        // 执行消息摘要
        byte[] b = encodeHmacRipeMD160(data, key);
        // 做十六进制转换
        return new String(Hex.encode(b));
    }
```

对上述代码做相关测试，如代码清单6-20所示。

代码清单6-20 HmacRipeMD系列算法实现测试用例

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * HmacRipeMD校验
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class HmacRipeMDCoderTest {
    /**
     * 测试HmacRipeMD128
     * @throws Exception
     */
    @Test
    public final void testEncodeHmacRipeMD128() throws Exception {
        String str = "HmacRipeMD128消息摘要";
        // 初始化密钥
        byte[] key = HmacRipeMDCoder.initHmacRipeMD128Key();
        // 获得摘要信息
```

更多资源请访问稀酷客(www.ckook.com)

```
        byte[] data1 = HmacRipeMDCoder.encodeHmacRipeMD128(str.getBytes(), key);
        byte[] data2 = HmacRipeMDCoder.encodeHmacRipeMD128(str.getBytes(), key);
        // 校验
        assertArrayEquals(data1, data2);
    }
    /**
     * 测试HmacRipeMD128Hex
     * @throws Exception
     */
    @Test
    public final void testEncodeHmacRipeMD128Hex() throws Exception {
        String str = "HmacRipeMD128Hex消息摘要";
        // 初始化密钥
        byte[] key = HmacRipeMDCoder.initHmacRipeMD128Key();
        // 获得摘要信息
        String data1 = HmacRipeMDCoder.encodeHmacRipeMD128Hex(str.getBytes(), key);
        String data2 = HmacRipeMDCoder.encodeHmacRipeMD128Hex(str.getBytes(), key);
        System.err.println("原文: \t" + str);
        System.err.println("HmacRipeMD128Hex-1: \t" + data1);
        System.err.println("HmacRipeMD128Hex-2: \t" + data2);
        // 校验
        assertEquals(data1, data2);
    }
    /**
     * 测试HmacRipeMD160
     * @throws Exception
     */
    @Test
    public final void testEncodeHmacRipeMD160() throws Exception {
        String str = "HmacRipeMD160消息摘要";
        // 初始化密钥
        byte[] key = HmacRipeMDCoder.initHmacRipeMD160Key();
        // 获得摘要信息
        byte[] data1 = HmacRipeMDCoder.encodeHmacRipeMD160(str.getBytes(), key);
        byte[] data2 = HmacRipeMDCoder.encodeHmacRipeMD160(str.getBytes(), key);
        // 校验
        assertArrayEquals(data1, data2);
    }
    /**
     * 测试HmacRipeMD160Hex
     * @throws Exception
     */
    @Test
    public final void testEncodeHmacMD4Hex() throws Exception {
        String str = "HmacRipeMD160Hex消息摘要";
        // 初始化密钥
        byte[] key = HmacRipeMDCoder.initHmacRipeMD160Key();
        // 获得摘要信息
```

更多资源请访问稀酷客(www.ckook.com)

```
            String data1 = HmacRipeMDCoder.encodeHmacRipeMD160Hex(str.getBytes(), key);
            String data2 = HmacRipeMDCoder.encodeHmacRipeMD160Hex(str.getBytes(), key);
            System.err.println("原文: \t" + str);
            System.err.println("HmacRipeMD160Hex-1: \t" + data1);
            System.err.println("HmacRipeMD160Hex-2: \t" + data2);
            // 校验
            assertEquals(data1, data2);
    }
}
```

以下是控制台中对应的输出信息：

```
原文:                    HmacRipeMD128Hex消息摘要
HmacRipeMD128Hex-1:    f08ccebbafe66f852ade9c73dcfdf14f
HmacRipeMD128Hex-2:    f08ccebbafe66f852ade9c73dcfdf14f
原文:                    HmacRipeMD160Hex消息摘要
HmacRipeMD160Hex-1:    0843ab8d7f3c3d0aca920fa7b8429268bacf5615
HmacRipeMD160Hex-2:    0843ab8d7f3c3d0aca920fa7b8429268bacf5615
```

消息摘要长度与相应的摘要算法的摘要长度相同：HmacRipeMD128与RipeMD128相对应，消息摘要都是32个字符的十六进制串；HmacRipeMD160与RipeMD160相对应，消息摘要值都是40个字符的十六进制串。

目前，Bouncy Castle不仅提供了HmacRipeMD系列算法实现，同时还提供了HMacTiger算法实现。实现方式与代码清单6-19相类似。

作者在整理Bouncy Castle加密组件时，才发现原来还有这么多不为人知的消息摘要算法。可见，各种消息摘要算法正以其独到之处快速地发展着，经过一番优胜劣汰，逐步成为标准。

6.6 循环冗余校验算法——CRC算法

大家对于“奇偶校验码”和“循环冗余校验码”这样的名词应该不会感到陌生。即便不是计算机专业出身，也一定使用过压缩软件WinRAR，在压缩文件列表中能看到一个标有“CRC32”字样的内容，如图6-8所示。

名称	大小	压缩后大小	类型	修改时间	CRC32
..			Folder		
local_policy.jar	2,481	2,142	Executable Jar File	2006/11/16 18...	045A71CF
US_export_policy.jar	2,465	2,127	Executable Jar File	2006/11/16 18...	8725E845
COPYRIGHT.html	2,663	1,244	Firefox Document	2006/11/16 18...	722D6301
README.txt	8,386	2,856	Text Document	2006/11/16 18...	E6110DE8

图6-8 压缩包中的CRC32算法

“奇偶校验码”、“循环冗余校验码”和“CRC32”都是同一套东西，它们和CRC有着紧密的联系。

CRC算法并不属于加密算法范畴，但与本章内容较为接近，故在这里为读者朋友做简要介绍。

更多资源请访问稀酷客(www.ckook.com)

6.6.1　简述

CRC（Cyclic Redundancy Check，循环冗余校验）是可以根据数据产生简短固定位数的一种散列函数，主要用来检测或校验数据传输/保存后出现的错误。生成的散列值在传输或储存之前计算出来并且附加到数据后面。在使用数据之前，对数据的完整性做校验。一般来说，循环冗余校验的值都是32位的二进制数，以8位十六进制字符串形式表示。它是一类重要的线性分组码，编码和解码方法简单，检错和纠错能力强，在通信领域广泛地用于实现差错控制。

由上述内容分析，消息摘要算法与CRC算法同属散列函数，并且CRC算法很可能就是消息摘要算法的前身。

CRC算法历经了多个版本的演进，从最初的CRC-1算法至最终的CRC-160算法，为消息摘要算法奠定了基础。我们简单回顾一下CRC算法的发展历程：

- CRC-1：主要用于硬件，就是我们常说的奇偶校验码。
- CRC-32-IEEE 802.3：主要用于通信领域实现差错控制，也就是我们今天常说的CRC-32，IEEE 802.3只是一种标准。
- CRC-32-Adler：CRC-32的一个变种，可以称为“Adler-32”，与CRC-32一样可靠，但是速度更快。
- CRC-128：演变为今天的MD算法。MD算法消息摘要值为128位二进制数。
- CRC-160：演变为今天的SHA算法。SHA-1算法消息摘要值为160位二进制数。

至今，CRC-32算法仍是各种压缩算法中最为常用的数据完整性校验算法。而它的变种Adler-32普遍用于zlib压缩算法中的数据完整性校验。

6.6.2　模型分析

我们以甲乙双方传递压缩数据模型为例，演示如何使用CRC算法。

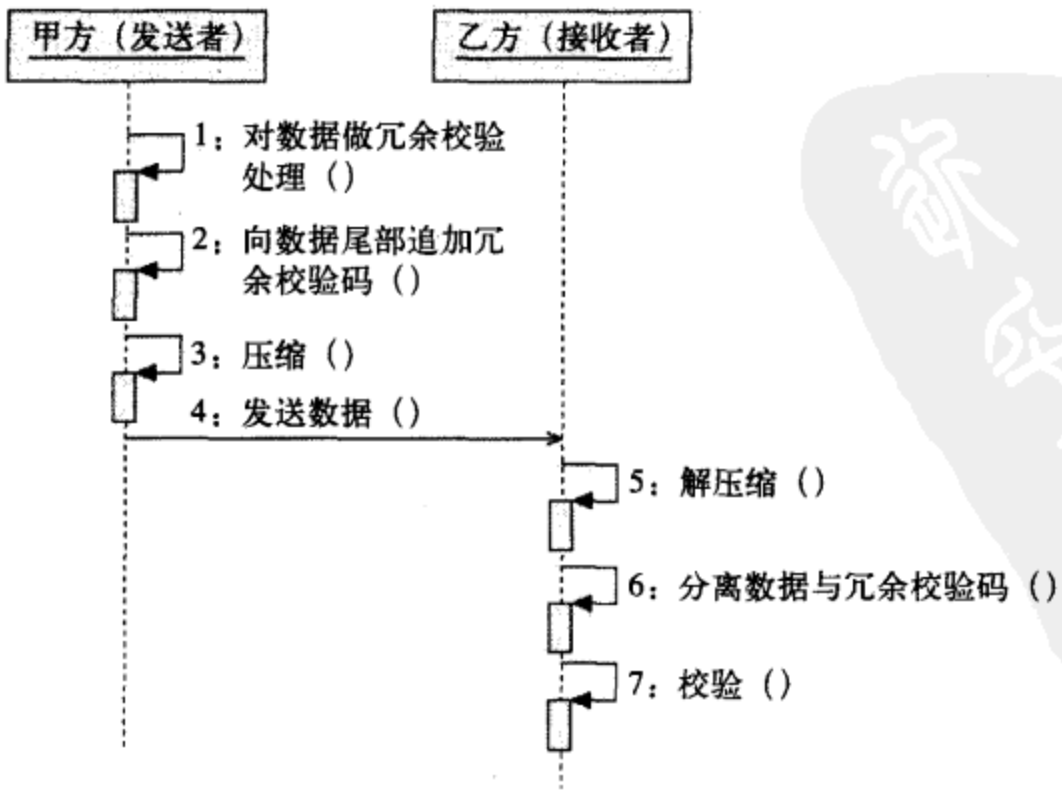


图6-9　压缩/解压缩数据传递模型

更多资源请访问稀酷客(www.ckook.com)

6.6.3 实现

在Java 6中，CRC-32算法是由CRC32类来实现的。

```
// 可用于计算数据流的CRC-32的类。
public class CRC32
extends Object
implements Checksum
```

CRC32类使用起来极为简单，通过其无参构造方法完成实例化对象后，需要先执行以下update()方法：

```
// 使用指定的字节数组更新校验和。
public void update(byte[] b)
// 使用指定的字节数组更新 CRC-32。
public void update(byte[] b, int off, int len)
// 使用指定字节更新 CRC-32。
public void update(int b)
```

update()方法可多次执行，此后执行getValue()方法：

```
// 返回 CRC-32 值。
public long getValue()
```

执行完上述方法后，我们可以获得一个长整型的冗余校验码，可通过Long类的toHexString()方法将其转换为十六进制字符串形式。

如需重置，需执行如下方法：

```
// 将 CRC-32 重置为初始值。
public void reset()
```

除此之外，还有CheckedInputStream和CheckedOutputStream两个类，可用于输入输出流的冗余校验处理。对于Adler-32算法，请使用Adler32类。Adler32类同样实现了Checksum接口，其方法与CRC32相同。

CRC32算法实现较为简单，如代码清单6-21所示。

代码清单6-21 CRC32算法实现

```
import java.util.zip.CRC32;
import org.junit.Test;
/**
 * 测试CRC-32
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class CRCTest {
    /**
     * 测试CRC-32
     * @throws Exception
```

更多资源请访问稀酷客(www.ckook.com)

```
     */
    @Test
    public void testCRC32() throws Exception {
            String str = "测试CRC-32";
            CRC32 crc32 = new CRC32();
            crc32.update(str.getBytes());
            String hex = Long.toHexString(crc32.getValue());
            System.err.println("原文: \t" + str);
            System.err.println("CRC-32: \t" + hex);
    }
}
```

执行上述代码后，在控制台中得到如下内容：

```
原文:       测试CRC-32
CRC-32:     eda03da1
```

原文经CRC-32算法处理后得到了一个8位十六进制字符串。

对上述代码稍作调整，就可用于文件校验。

6.7 实例：文件校验

秉承程序员的执著，发现问题就要解决问题！既然我们已经看到了MySQL下载页面上的MD5信息，就要对其文件做MD5校验！通过下载页面下载MySQL Essential 5.1.38的Windows安装版（mysql-essential-5.1.38-win32.msi），并将其放置在D盘根目录下。接下来，我们要对这个文件做一次MD5校验。如果校验结果与下载页面上给出的MD5信息不一致，则说明文件在下载的过程中被篡改了！

这个校验并不复杂，我们分别通过testByMessageDigest()和testByDigestUtils()方法做校验处理，如代码清单6-22所示。

代码清单6-22 校验下载文件一致性

```
import static org.junit.Assert.*;
import java.io.File;
import java.io.FileInputStream;
import java.security.DigestInputStream;
import java.security.MessageDigest;
import org.apache.commons.codec.binary.Hex;
import org.apache.commons.codec.digest.DigestUtils;
import org.junit.Test;
/**
 * 消息摘要编码测试<br>
 * 用于校验文件的MD5值
 * <pre>
 * 文件为 mysql-essential-5.1.38-win32.msi
 * 存放于D盘根目录
```

更多资源请访问稀酷客(www.ckook.com)

```
 * MD5值为5a077abefee447cbb271e2aa7f6d5a47
 * </pre>
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class MD5Test {
    /**
     * 验证文件的MD5值
     * @throws Exception
     */
    @Test
    public void testByMessageDigest() throws Exception {
            // 文件路径
            String path = "D:\\mysql-essential-5.1.38-win32.msi";
            // 构建文件输入流
            FileInputStream fis = new FileInputStream(new File(path));
            // 初始化MessageDigest，并指定MD5算法
            DigestInputStream dis = new DigestInputStream(fis, MessageDigest.getInstance("MD5"));
            // 流缓冲大小
            int buf = 1024;
            // 缓冲字节数组
            byte[] buffer = new byte[buf];
            // 当读到值大于-1就继续读
            int read = dis.read(buffer, 0, buf);
            while (read > -1) {
                read = dis.read(buffer, 0, buf);
            }
            // 关闭流
            dis.close();
            // 获得MessageDigest
            MessageDigest md = dis.getMessageDigest();
            // 摘要处理
            byte[] b = md.digest();
            // 十六进制转换
            String md5hex = Hex.encodeHexString(b);
            // 验证
            assertEquals(md5hex, "5a077abefee447cbb271e2aa7f6d5a47");
}
/**
 * 验证文件的MD5值
 * @throws Exception
 */
@Test
public void testByDigestUtils() throws Exception {
            // 文件路径
```

更多资源请访问稀酷客(www.ckook.com)

```
        String path = "D:\\mysql-essential-5.1.38-win32.msi";
        // 构建文件输入流
        FileInputStream fis = new FileInputStream(new File(path));
        // 使用DigestUtils做MD5Hex处理
        String md5hex = DigestUtils.md5Hex(fis);
        // 关闭流
        fis.close();
        // 验证
        assertEquals(md5hex, "5a077abefee447cbb271e2aa7f6d5a47");
    }
}
```

既然两种方法都是MD5文件校验，那么会有什么差异呢？

从测试结果来说，一定都是通过，那么在效率上呢？如图6-10所示。

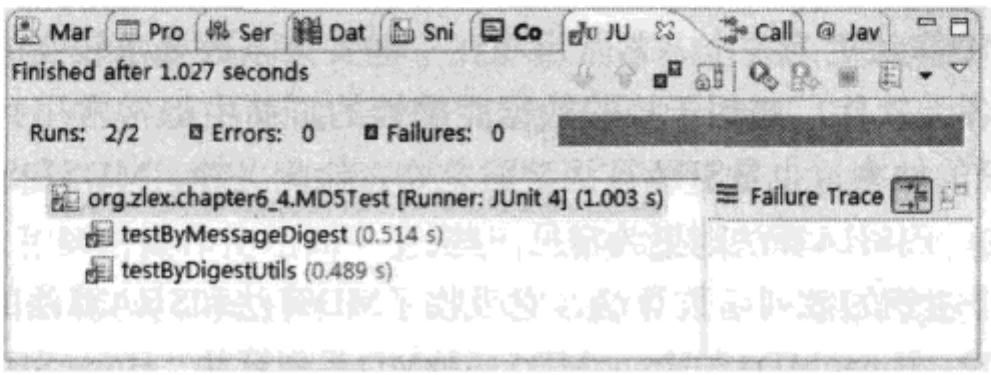


图6-10 MD5文件校验处理测试结果

作者做了反复测试，其测试结果与上图基本相符，两种方法实现效率较为接近。

如果只是对文件做MD5校验，用Commons Codec来实现再合适不过了。对文件做MD5处理/校验，已经是一件很平常的事情了。相信上述简要的代码实现，对读者朋友一定会很有益处。

6.8 小结

对于数据完整性的验证，可能是每一个计算机用户所关心的事情。这好比是自己在菜市场上买了菜要验分量。但如何来验证数据是否完整，而且是快速准确地验证数据的完整性又是一个麻烦的问题，这需要一种算法。

消息摘要算法就是这样一种专门用于验证数据完整性的算法。它源于CRC冗余校验算法，派生出MD和SHA两大系列消息算法，并在此基础上衍生出MAC算法。消息摘要算法是数字签名算法的基础。

几乎每一种消息摘要算法都有大概三种实现方式：Sun、Bouncy Castle和Commons Codec。Sun提供了最基本的算法实现，Bouncy Castle在Sun的基础上做了扩展，实现了Sun未能提供的算法；Commons Codec在Sun的基础上，对其方法做了封装，使其易用性提高，方便使用。

MD算法家族拥有MD2、MD4和MD5三种常用算法。在Java 6中，通过MessageDigest类可提供MD2和MD5两种算法支持；通过Bouncy Castle扩展，可提供MD4算法支持；Commons Codec则直接强化了MD5算法使用的便利性。如果考虑MD4算法的支持，则使用Bouncy

更多资源请访问稀酷客(www.ckook.com)

Castle；如果考虑方法的易用性，则使用Commons Codec更为恰当。

虽然MD5算法的破解使其安全性大为降低，但在用户注册/登录模块中仍然是架构师首选的方案。各大软件厂商在其软件下载页面上仍然使用MD5算法作为数据完整性验证的首选方法。MD5算法常作为安全性要求不高的环境中的常用算法。MD4和MD5算法为后续消息摘要算法（如SHA）的设计提供了参考。

SHA算法家族枝繁叶茂，拥有SHA-1、SHA-224、SHA-256、SHA-384和SHA-512五种常用算法。其中，后四种算法是在原有SHA-1基础上扩展了消息摘要长度，通常也称为SHA-2算法。在Java 6中，通过MessageDigest类可提供SHA-1、SHA-256、SHA-384和SHA-512四种算法支持；通过Bouncy Castle扩展，可提供SHA-224算法支持；Commons Codec则强化了SHA-1、SHA-256、SHA-384和SHA-512四种算法相应方法的易用性。如果不要求使用SHA-224算法，使用Commons Codec更方便。

SHA算法较之MD算法更为安全，常常出现在一些安全系数要求较高的环境中。一般的用户注册/登录模块，各大软件厂商用于校验数据完整性的页面中都常常用到SHA算法。这些领域既是MD5算法出没的地方，也是SHA算法盘踞之处。除此之外，MD5和SHA算法还常常作为数字证书的签名算法，而SHA算法则更为常见一些。

MAC是一种基于密钥的散列函数算法，它吸收了MD算法和SHA算法的精髓，并将其发扬光大，包含HmacMD2、HmacMD4和HmacMD5三种MD系列算法，HmacSHA1、HmacSHA224、HmacSHA256、HmacSHA384和HmacSHA512五种SHA系列算法。在Java 6中，通过Mac类可以提供HmacMD5、HmacSHA1、HmacSHA256、HmacSHA384和HmacSHA512五种算法支持；通过Bouncy Castle扩展，可提供HmacMD2、HmacMD4和HmacSHA224三种算法支持。如果你想让你的系统更强劲，使用Bouncy Castle就是最好的选择。

虽然已经有了主流的消息摘要算法实现，但仍可能不能满足我们的需要。非主流消息摘要算法RipeMD系列（括含RipeMD128、RipeMD160、RipeMD256和RipeMD320）、Tiger、Whirlpool和GOST3411算法，以及HmacRipeMD系列（包括HmacRipeMD128和HmacRipeMD160）算法，可通过Bouncy Castle完成实现。

MD、SHA和MAC都是加密算法领域的消息摘要算法，与之功能相近的CRC-32算法则是最为古老的数据完整性验证算法。目前，CRC-32算法仍广泛用于通信领域，实现差错控制，其变种Adler-32算法则广泛适用于zlib压缩算法中。

更多资源请访问稀酷客(www.ckook.com)

第7章

初等数据加密——对称加密算法

我们都有使用密码保护私密信息的经历，甚至可以说是习惯。我们往往不希望无关的人窥探我们的隐私，从孩童时代就知道用“密码日记本”记录自己的一些隐私。密码日记本无非是一个带锁的日记本。不管是读日记还是写日记都离不开这个密码。

上述情形就好比我们应用黑匣子，需要读写操作，需要同一套密钥。写操作伴随加密，读操作伴随解密。加密和解密操作使用同一套密钥，这就是对称加密算法的核心。

对称加密算法在Java实现层面上大同小异，本文挑选了几个具有代表性的算法做相应实现。

7.1 对称加密算法简述

对称加密算法是当今应用范围最广，使用频率最高的加密算法。它不仅应用于软件行业，在硬件行业同样流行。各种基础设施但凡涉及安全需求，都会优先考虑对称加密算法。

7.1.1 对称加密算法的由来

对于大多数对称加密算法而言，解密算法是加密算法的逆运算，加密密钥和解密密钥相同。如果我们把Base64算法改良，将其字符映射表作为密钥保存，就可以把这个改良后的Base64算法作为一种对称加密算法来看。当然，加密算法这样改良后的安全强度还远远不够，但足以让我们认识对称加密算法的特点。

对称加密算法易于理解，便于实现，根据加密方式又分为密码和分组密码，其分组密码工作模式又可分为ECB、CBC、CFB、OFB和CTR等，密钥长度决定了加密算法的安全性。有关对称加密算法相关理论知识，请阅读第2章相关内容。

对称加密算法发展至今已相当完备。以DES算法为例，由于密钥长度的不满足，衍生出了DESede算法（也称为TripleDES或3DES算法，翻译成中文是“三重DES”算法）。为了替代DES算法又有了AES（Rijndael）算法。此外，还有RC系列算法，包含RC2、RC4以及针对32位/64位计算机设计的RC5算法（细分为RC5-32和RC5-64，分别对应32位和64位计算机）。

除了上述算法，我们还常常会用到Blowfish、Twofish、Serpent、IDEA和PBE等对称加密算法。

更多资源请访问稀酷客(www.ckook.com)

7.1.2 对称加密算法的家谱

目前已知的可通过Java语言实现的对称加密算法大约有20多种，Java 6仅仅提供了部分算法实现，如DES、DESede、AES、Blowfish，以及RC2和RC4算法等。其他算法（如IDEA算法）需要通过第三方加密软件包Bouncy Castle提供实现。

在对称加密算法中，DES算法最具有代表性，堪称典范；DESede是DES算法的变种；AES算法则作为DES算法的替代者；而IDEA算法作为一种强加密算法，成为电子邮件加密软件PGP（Pretty Good Privacy）的核心算法之一。

在Java实现层面上，DES、DESede、AES和IDEA这4种算法略有不同。

DES和DESede算法在使用密钥材料还原密钥时，建议使用各自相应的密钥材料实现类（DES算法对应DESKeySpec类，DESede算法对应DESedeKeySpec类）完成相应转换操作。

AES算法在使用密钥材料还原密钥时，则需要使用一般密钥材料实现类（SecretKeySpec类）完成相应转换操作。其他对称加密算法可参照该方式实现，如RC2、RC4、Blowfish以及IDEA等算法均可参照AES算法实现方式做相应实现。

IDEA算法实现Java 6未能提供，需要依赖第三方加密组件包Bouncy Castle提供支持，其他由Bouncy Castle提供支持的对称加密算法可参照该算法实现方式做相应实现。

7.2 数据加密标准——DES

DES算法和DESede算法统称DES系列算法。DES算法是对称加密算法领域中的典型算法，为后续对称加密算法的发展奠定了坚实的基础。DESede算法基于DES算法进行三重迭代，增加其算法安全性。经过一番筛选，Rijndeal算法最终成为了AES算法。这期间，对称加密算法发展迅速，与Rijndeal算法竞争的算法包括Blowfish、Serpent等。IDEA算法也源于增加算法的安全性替代DES算法，诸多对称加密算法的发展均源于DES算法的研究而来。

7.2.1 简述

1973年，美国国家标准局（NBS，National Bureau of Standards）（即现在的美国国家标准技术研究所（National Institute of Standards and Technology，NIST））征求国家密码标准方案，IBM公司提交了自己研制的算法（Luciffer算法，于1971年末提出）。1977年7月15日，该算法被正式采纳，作为美国联邦信息处理标准生效，并很快应用到国际商用数据加密领域，成为事实标准，即数据加密标准（Data Encryption Standard，DES），DES算法由此诞生。

DES算法作为现代密码学领域中第一个官方授权的加密算法受到全球各大密码学机构的关注。DES算法密钥偏短，仅有56位，迭代次数偏少，受到诸如查分密码分析和线性密码分析等各种攻击威胁，安全性受到严重威胁。不仅如此，由于DES算法具有半公开性质，被怀疑存在美国国家安全局（National Security Agency，NSA）安置的后门，受到各大密码学机构的强烈质疑。

更多资源请访问稀酷客(www.ckook.com)

1998年后，实用化DES算法破译机的出现彻底宣告DES算法已不具备安全性。1999年NIST颁布新标准，规定DES算法只能用于遗留加密系统，但不限制使用DESede算法。以当今计算机技术能力，经DES算法加密的数据在24小时内可能被破解。由此，DES算法正式退出历史舞台，AES算法成为它的替代者。

即便如此，DES算法对于密码学领域的贡献确实是巨大的。各种对称加密算法均由研究DES算法发展而来，对后续对称加密算法的发展起到奠基作用。DES算法实现不仅遍布软件行业，甚至很多硬件芯片本身也具备DES加密实现。同时，作为一款较易实现的加密算法，DES算法也成为最应学习的对称加密算法，其地位堪比C语言在计算机语言中的地位。

基于CBC工作模式的DES算法相关文档可参考RFC 1829（http://www.ietf.org/rfc/rfc1829.txt）。

历经20年发展，DES算法不仅应用在软件行业，成为电子商务必不可少的加密算法，同时也逐步渗透到硬件行业。芯片级DES算法生产工艺相当完备，完全可以支持底层加密需求。

7.2.2　模型分析

第5章曾介绍过基于Base64算法的消息交互模型，如图7-1所示。

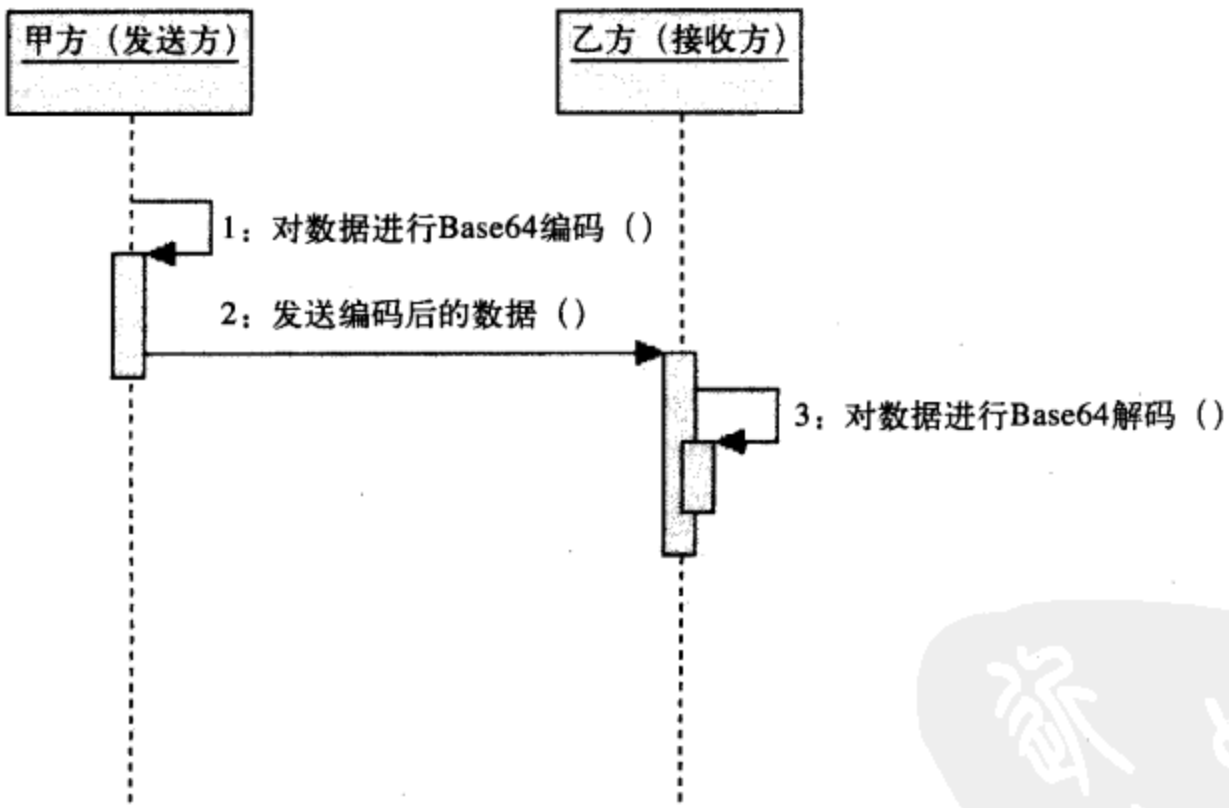


图7-1　基于Base64算法的消息交互模型

对于对称加密算法，其消息模型并没有太大的差别。基于DES算法的消息传递模型如图7-2所示。

两幅时序图的主要差别在于密钥的使用。基于Base64算法的消息传递模型中没有密钥的概念。那是因为Base64字符映射表本身已经公开，而Base64字符映射表本身起到了密钥的作用。消息传递双方通讯前不需要商榷该密钥，也就省去了构建密钥、公布密钥的步骤。

甲乙双方作为消息传递双方（甲方作为发送方，乙方作为接收方），我们假定甲乙双方在消息传递前已商定加密算法，欲完成一次消息传递需经过如下步骤：

更多资源请访问稀酷客(www.ckook.com)

1）由消息传递双方约定密钥，这里由甲方构建密钥。
2）由密钥构建者公布密钥，这里由甲方将密钥公布给乙方。
3）由消息发送方使用密钥对数据加密，这里由甲方对数据加密。
4）由消息发送方将加密数据发送给消息接收者，这里由甲方将加密数据发送给乙方。
5）由消息接收方使用密钥对加密数据解密，这里由乙方完成数据解密。

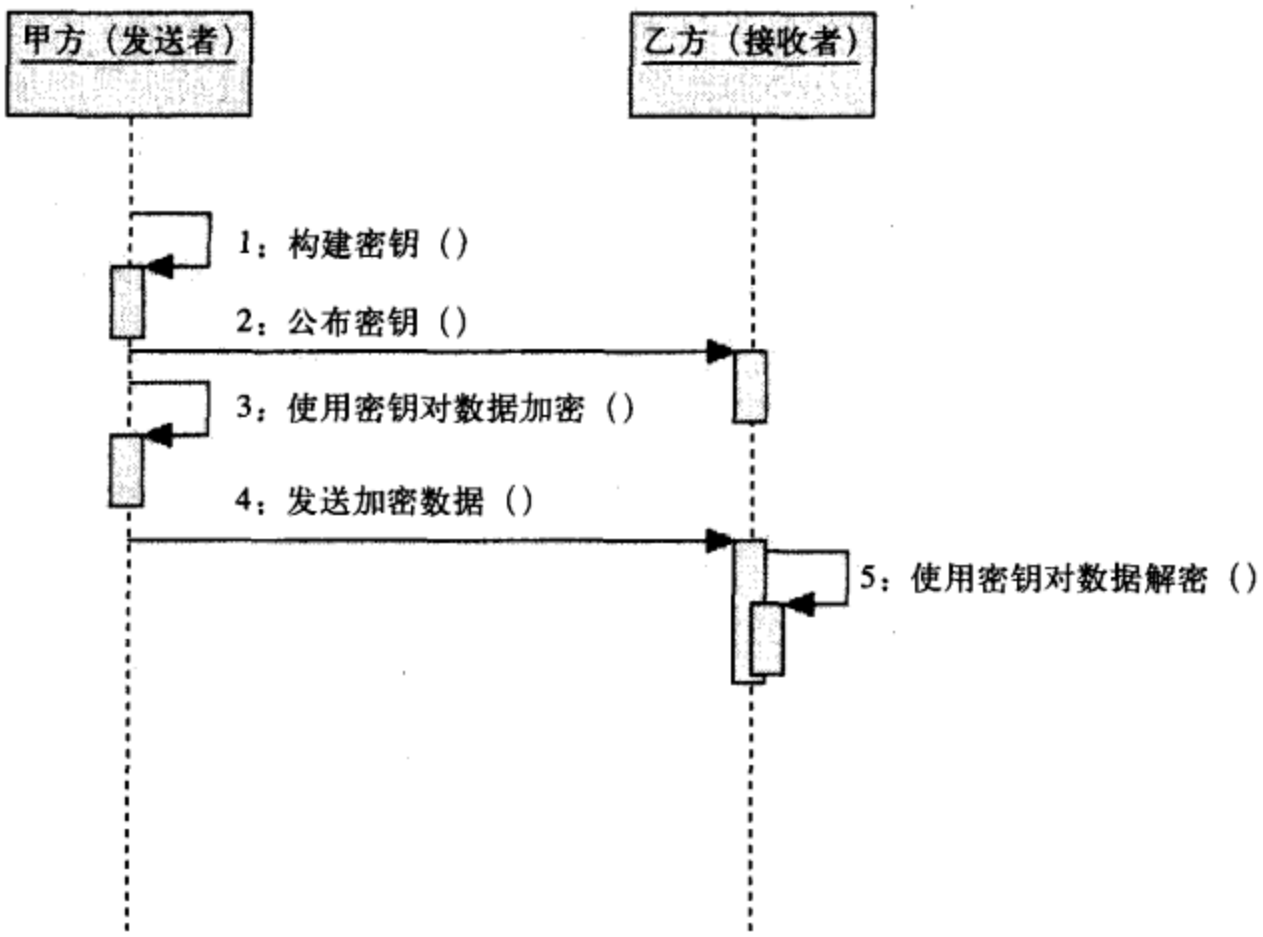


图7-2 基于DES算法的消息传递模型

对称加密算法的优点就是简单易行，通俗易懂。对于上述单向消息传递而言，如果乙方想要回复甲方消息，并不需要重复步骤1、2，仅仅须由乙方执行步骤3、4，由甲方执行步骤5即可。

基于DES算法构建的消息传递模型大都按照上述模型构建，同时包括其他对称加密算法，如DESede、AES和IDEA等。当然，并不是所有基于对称加密算法的消息传递模型都按此步骤构建，PBE算法就是一个例外。我们将在后续内容中单独详述基于PBE算法的消息传递模型。

7.2.3 实现

我们知道，密钥长度与安全性成正比，但Java 6仅支持56位密钥长度，作为补充，Bouncy Castle提供64位密钥长度支持。在此基础上配合不同的填充方式（如PKCS5Padding，PKCS7Padding），可显著提高加密系统的安全性。

有关DES算法的Java 6实现与Bouncy Castle实现细节如表7-1所示。

更多资源请访问稀酷客(www.ckook.com)

表7-1 DES算法

算法	密钥长度	密钥长度默认值	工作模式	填充方式	备注
DES	56	56	ECB、CBC、PCBC、CTR、CTS、CFB、CFB8至CFB128、OFB、OFB8至OFB128	NoPadding、PKCS5Padding、ISO10126Padding	Java 6实现
	64	同上	同上	PKCS7Padding、ISO10126d2Padding、X932Padding、ISO7816d4Padding、ZeroBytePadding	Bouncy Castle实现

请读者朋友注意DES算法的实现过程，其他对称加密算法与该算法实现相类似。

密钥的构建主要需要密钥生成器（KeyGenerator）完成生成操作，如下代码所示：

```
// 实例化密钥生成器
KeyGenerator kg = KeyGenerator.getInstance("DES");
// 初始化
kg.init(56);
// 生成秘密密钥
SecretKey secretKey = kg.generateKey();
// 获得密钥的二进制编码形式
byte[] b = secretKey.getEncoded();
```

字节数组b就是我们需要的秘密密钥字节数组形式。这便于我们将其存储在文件中，或以数据流的形式在网络中传输。

把密钥转化为二进制字节数组形式便于保存，但若我们要使用它需要将其转换为密钥对象，首先需要将二进制密钥转换为密钥材料对象（这里是DESKeySpec对象dks），再使用密钥工厂（SecretKeyFactory）生成密钥。实现代码如下所示：

```
// 实例化DES密钥材料
DESKeySpec dks = new DESKeySpec(b);
// 实例化秘密密钥工厂
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES");
// 生成秘密密钥
SecretKey secretKey = keyFactory.generateSecret(dks);
```

对于DESede算法，则需要相应的DESedeKeySpec类替换DESKeySpec类来完成上述操作。

得到密钥对象后，我们就可以对数据做加密/解密处理，加密处理代码如下所示：

```
// 实例化
Cipher cipher = Cipher.getInstance("DES");
// 初始化，设置为加密模式
cipher.init(Cipher.ENCRYPT_MODE, secretKey);
// 执行操作
byte[] data = cipher.doFinal(data);
```

更多资源请访问稀酷客(www.ckook.com)

上述实例化操作未指定工作模式及填充方式，我们将在后续内容中详述。

如果将上述初始化方法（init()方法）中的模式参数由“Cipher.ENCRYPT_MODE”改为“Cipher.DECRYPT_MODE”，则可作为解密处理。

Java 6提供了DES算法支持，但仅支持56位的密钥长度。我们知道密钥长度与加密强度成正比。我们可以使用Boucy Calstle提供密钥长度，由56位提高至64位。

接下来我们完成一套基于DES算法的密钥构建和加密/解密操作，如代码清单7-1所示。

代码清单7-1 DES算法实现

```
import java.security.Key;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.DESKeySpec;
/**
 * DES安全编码组件
 * @author 梁栋
 * @version 1.0
 */
public abstract class DESCoder {
    /**
     * 密钥算法 <br>
     * Java 6 只支持56位密钥 <br>
     * Bouncy Castle 支持64位密钥
     */
    public static final String KEY_ALGORITHM = "DES";
    /**
     * 加密/解密算法 / 工作模式 / 填充方式
     */
    public static final String CIPHER_ALGORITHM = "DES/ECB/PKCS5Padding";
    /**
     * 转换密钥
     * @param key 二进制密钥
     * @return Key 密钥
     * @throws Exception
     */
    private static Key toKey(byte[] key) throws Exception {
            // 实例化DES密钥材料
            DESKeySpec dks = new DESKeySpec(key);
            // 实例化秘密密钥工厂
            SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(KEY_ALGORITHM);
            // 生成秘密密钥
            SecretKey secretKey = keyFactory.generateSecret(dks);
            return secretKey;
    }
    /**
```

更多资源请访问稀酷客(www.ckook.com)

```
     * 解密
     * @param data 待解密数据
     * @param key 密钥
     * @return byte[] 解密数据
     * @throws Exception
     */
    public static byte[] decrypt(byte[] data, byte[] key) throws Exception {
            // 还原密钥
            Key k = toKey(key);
            // 实例化
            Cipher cipher = Cipher.getInstance(CIPHER_ALGORITHM);
            // 初始化，设置为解密模式
            cipher.init(Cipher.DECRYPT_MODE, k);
            // 执行操作
            return cipher.doFinal(data);
    }
    /**
     * 加密
     * @param data 待加密数据
     * @param key 密钥
     * @return byte[] 加密数据
     * @throws Exception
     */
    public static byte[] encrypt(byte[] data, byte[] key) throws Exception {
            // 还原密钥
            Key k = toKey(key);
            // 实例化
            Cipher cipher = Cipher.getInstance(CIPHER_ALGORITHM);
            // 初始化，设置为加密模式
            cipher.init(Cipher.ENCRYPT_MODE, k);
            // 执行操作
            return cipher.doFinal(data);
    }
    /**
     * 生成密钥 <br>
     * Java 6 只支持56位密钥 <br>
     * Bouncy Castle 支持64位密钥 <br>
     * @return byte[] 二进制密钥
     * @throws Exception
     */
    public static byte[] initKey() throws Exception {
            /*
             * 实例化密钥生成器
             * 若要使用64位密钥注意替换
             * 将下述代码中的
             * KeyGenerator.getInstance(CIPHER_ALGORITHM);
```

更多资源请访问稀酷客(www.ckook.com)

```
            * 替换为
            * KeyGenerator.getInstance(CIPHER_ALGORITHM, "BC");
            */
        KeyGenerator kg = KeyGenerator.getInstance(KEY_ALGORITHM);
        /*
         * 初始化密钥生成器
         * 若要使用64位密钥注意替换
         * 将下述代码kg.init(56);
         * 替换为kg.init(64);
         */
        kg.init(56);
        // 生成秘密密钥
        SecretKey secretKey = kg.generateKey();
        // 获得密钥的二进制编码形式
        return secretKey.getEncoded();
    }
}
```

请读者朋友注意上述代码中的生成密钥方法（initKey()），如果我们初始化密钥生成器时按如下方式实现，将获得一个默认长度的密钥：

```
kg.init(new SecureRandom());
```

Java 6仅仅提供了56位长度的密钥，因此上述方法将产生一个56位长度的密钥。

若我们想要构建一个64位密钥的DES算法，则需要按如下代码做替换实现：

```
KeyGenerator kg = KeyGenerator.getInstance(CIPHER_ALGORITHM, "BC");
```

在上述代码中，“BC”是Boucy Calstle安全提供者的缩写。

当然，你也可以使用如下方式替代上述代码：

```
import java.security.Security;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
// 省略
// 加入BouncyCastleProvider支持
Security.addProvider(new BouncyCastleProvider());
KeyGenerator kg = KeyGenerator.getInstance(CIPHER_ALGORITHM);
```

完成密钥生成器实例化操作后，需要注意密钥生成器初始化操作，如以下代码所示：

```
kg.init(64);
```

按上述代码实现方式，我们即可获得Boucy Calstle安全提供者提供的64位的DES算法密钥。

注意 密钥生成和加密/解密所使用的算法很可能是不同的。

在本文的DES算法实现中，密钥算法（变量KEY_ALGORITHM）是“DES”，而加密/解密算法（变量“CIPHER_ALGORITHM”）是“DES/ECB/PKCS5Padding”。这里的加密/解密算法中除了包含密钥算法（DES）外，还包含了工作模式（ECB）和填充方式（PKCS5Padding）。

如果密钥算法与加密/解密算法一致，则按默认工作模式和填充方式实现。

更多资源请访问稀酷客(www.ckook.com)

在实际应用中，密文通常以二进制数据传输/存储，而密钥通常会被转换为可见字符存储。如使用Base64编码或十六进制编码将不可见的二进制密钥转换为可见字符。当然，这里需要注意一点，若使用Base64编码，则编码后的信息将是原始信息长度的4/3倍（相关原理请阅读第5章）。

为了便于演示，作者将密文和密钥均使用Base64编码形式展示（我们使用Commons Codec完成Base64算法实现，相关内容请参考第5章）。测试用例实现如代码清单7-2所示。

代码清单7-2 DES算法实现测试用例

```
import static org.junit.Assert.*;
import org.apache.commons.codec.binary.Base64;
import org.junit.Test;
/**
 * DES安全编码组件校验
 * @author 梁栋
 * @version 1.0
 */
public class DESCoderTest {
    /**
     * 测试
     * @throws Exception
     */
    @Test
    public final void test() throws Exception {
            String inputStr = "DES";
            byte[] inputData = inputStr.getBytes();
            System.err.println("原文:\t" + inputStr);
            // 初始化密钥
            byte[] key = DESCoder.initKey();
            System.err.println("密钥:\t" + Base64.encodeBase64String(key));
            // 加密
            inputData = DESCoder.encrypt(inputData, key);
            System.err.println("加密后:\t" + Base64.encodeBase64String(inputData));
            // 解密
            byte[] outputData = DESCoder.decrypt(inputData, key);
            String outputStr = new String(outputData);
            System.err.println("解密后:\t" + outputStr);
            // 校验
            assertEquals(inputStr, outputStr);
    }
}
```

我们来观察控制台的输出信息，如下所示：

```
原文:       DES
密钥:       qA2oZBaKNOk=
```

更多资源请访问稀酷客(www.ckook.com)

```
加密后：    QwCjNM5G8KM=
解密后：    DES
```

加密/解密操作顺利完成！

这样的密钥短小精悍，便于记忆！作者与合作公司在构建加密通信模块时，通常会指定这样便于书写的密钥，并将其书写在合同书上发送给对方。这就是为什么要将密钥存储为Base64编码或十六进制编码的原因。

随着计算机的发展，密钥长度仅有56位的DES算法显得越来越不安全，虽然通过Bouncy Castle可将密钥长度增至64位，提高了其安全强度，但DES算法在设计上的漏洞已经不能通过单纯地增加密钥长度来弥补，这引发了对称加密算法研发竞赛！

DESede和AES算法正是这场竞赛中具有代表性的算法。

7.3 三重DES——DESede

作为DES算法的一种改良，DESede算法针对其密钥长度偏短和迭代次数偏少等问题做了相应改进，提高了安全强度。但这仍不是终点，DESede算法的出现仅为DES算法的改良提供了一种参考。DESede算法处理速度较慢，密钥计算时间较长，加密效率不高等问题使得对称加密算法的发展仍不容乐观。

7.3.1 简述

DES算法被广大密码学机构质疑的原因主要在于DES算法的半公开性，违反了柯克霍夫原则，各大密码学结构怀疑美国国家安全局在未公开的算法实现内安置后门。

DES算法有3点安全隐患：密钥太短、迭代偏少和半公开性。这使得淘汰DES算法成为一种必然，但要淘汰DES算法必须找到合适的替代方案。

针对密钥太短和迭代偏少问题，有人提出了多重DES的方式来克服这些缺陷。比较典型的有两重DES（2DES）、三重DES（3DES）和四重DES（4DES）等几种形式，但在实际应用中一般采用3DES方案，它还有两个别名Triple DES和DESede。在Java中，我们通常称其为DESede算法。当然，其他两种名称在使用时同样可以获得支持。

DESede算法将密钥长度增至112位或168位，抗穷举攻击的的能力显著增强，但核心仍是DES算法，虽然通过增加迭代次数提高了安全性，但同时也造成处理速度较慢，密钥计算时间加长，加密效率不高的问题。

7.3.2 实现

对DES算法实现有一定了解后，DESede算法的实现就容易许多。除了将密钥材料实现类由DESKeySpec类转换为DESedeKeySpec类外，DESede算法与DES算法实现的主要差别在于算法、密钥长度两个方面：

- **算法**：自然不用说，这是基本差别，只是DESede还有很多别名，如TripleDES和3DES指

更多资源请访问稀酷客(www.ckook.com)

的都是DESede算法。

- 密钥长度：Java 6提供的DES算法实现支持56位密钥长度，加上Bouncy Castle相应实现可以支持到64位密钥长度。Java 6提供的DESede算法实现所支持的密钥长度支持为112位和168位，通过Bouncy Castle相应实现可支持密钥长度为128位和192位。DESede算法密钥长度恰恰是DES算法密钥长度的2倍或3倍。

有关DESede算法的Java 6实现与Bouncy Castle实现细节如表7-2所示。

表7-2 DESede算法

算法	密钥长度	密钥长度默认值	工作模式	填充方式	备注
DESede（Triple DES，3DES）	112、168	168	ECB、CBC、PCBC、CTR、CTS、CFB、CFB8至CFB128、OFB、OFB8至OFB128	NoPadding、PKCS5Padding、ISO10126Padding	Java 6实现
	128、192	同上	同上	PKCS7Padding、ISO10126d2Padding、X932Padding、ISO7816d4Padding、ZeroBytePadding	Bouncy Castle实现

通过7.2节的DES算法实现演示，相信读者对于如何实现DES算法，以及如何使用Bouncy Castle扩充密钥长度已经很了解了，本文将阐述如何使用Bouncy Castle扩充填充方式。

对于DESede算法的填充方式，Java 6提供了NoPadding、PKCS5Padding和ISO10126Padding共3种填充方式。

作者与合作方商定加密算法时，对方技术水准较高，要求使用PKCS7Padding填充方式，着实令作者犯难，好在这时作者发现了Bouncy Castle，才解决了这一技术难题。仔细研究，发现Bouncy Castle不仅支持PKCS7Padding这一种填充方式，还支持ISO10126d2Padding、X932Padding、ISO7816d4Padding和ZeroBytePadding共4种填充方式。

对于如何使用PKCS7Padding填充方式完成DESede算法构建的问题，我们在代码清单7-3中详述。

代码清单7-3 DESede算法实现

```
import java.security.Key;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.DESedeKeySpec;
/**
 * DESede安全编码组件
 * @author 梁栋
 * @version 1.0
 */
```

更多资源请访问稀酷客(www.ckook.com)

```
public abstract class DESedeCoder {
    /**
     * 密钥算法
     * Java 6支持密钥长度为112位和168位
     * Bouncy Castle支持密钥长度为128位和192位
     */
    public static final String KEY_ALGORITHM = "DESede";
    /**
     * 加密/解密算法 / 工作模式 / 填充方式
     * Java 6支持PKCS5Padding填充方式
     * Bouncy Castle支持PKCS7Padding填充方式
     */
    public static final String CIPHER_ALGORITHM = "DESede/ECB/PKCS5Padding";
    /**
     * 转换密钥
     * @param key 二进制密钥
     * @return Key 密钥
     * @throws Exception
     */
    private static Key toKey(byte[] key) throws Exception {
            // 实例化DES密钥材料
            DESedeKeySpec dks = new DESedeKeySpec(key);
            // 实例化秘密密钥工厂
            SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(KEY_ALGORITHM);
            // 生成秘密密钥
            return keyFactory.generateSecret(dks);
    }
    /**
     * 解密
     * @param data 待解密数据
     * @param key 密钥
     * @return byte[] 解密数据
     * @throws Exception
     */
    public static byte[] decrypt(byte[] data, byte[] key) throws Exception {
            // 还原密钥
            Key k = toKey(key);
            /*
             * 实例化
             * 使用PKCS7Padding填充方式，按如下代码实现
             * Cipher.getInstance(CIPHER_ALGORITHM, "BC");
             */
            Cipher cipher = Cipher.getInstance(CIPHER_ALGORITHM);
            // 初始化，设置为解密模式
            cipher.init(Cipher.DECRYPT_MODE, k);
            // 执行操作
            return cipher.doFinal(data);
    }
```

更多资源请访问稀酷客(www.ckook.com)

```
/**
 * 加密
 * @param data 待加密数据
 * @param key 密钥
 * @return byte[] 加密数据
 * @throws Exception
 */
public static byte[] encrypt(byte[] data, byte[] key) throws Exception {
        // 还原密钥
        Key k = toKey(key);
        /*
         * 实例化
         * 使用PKCS7Padding填充方式，按如下代码实现
         * Cipher.getInstance(CIPHER_ALGORITHM, "BC");
         */
        Cipher cipher = Cipher.getInstance(CIPHER_ALGORITHM);
        // 初始化，设置为加密模式
        cipher.init(Cipher.ENCRYPT_MODE, k);
        // 执行操作
        return cipher.doFinal(data);
}
/**
 * 生成密钥 <br>
 * @return byte[] 二进制密钥
 * @throws Exception
 */
    public static byte[] initKey() throws Exception {
            /*
             * 实例化
             * 使用128位或192位长度密钥，按如下代码实现
             * KeyGenerator.getInstance(KEY_ALGORITHM, "BC");
             */
            KeyGenerator kg = KeyGenerator.getInstance(KEY_ALGORITHM);
            /*
             * 初始化
             * Java 6支持密钥长度为 112位和168位
             * 若使用128位或192位长度密钥，按如下代码实现
             * kg.init(128);
             * 或
             * kg.init(192);
             */
            kg.init(168);
            // 生成秘密密钥
            SecretKey secretKey = kg.generateKey();
            // 获得密钥的二进制编码形式
            return secretKey.getEncoded();
    }
}
```

更多资源请访问稀酷客(www.ckook.com)

上述代码与DES算法实现如出一辙。这要感谢Sun提供的JCE架构，它提供了统一的加密算法调用模式。我们注意到，这里的密钥材料实现类由DESKeySpec类改为DESedeKeySpec类，这是为DESede算法量身定做的密钥材料实现类。

除了密钥材料实现类的变化，还要注意密钥长度的区别。

Java 6支持112位和168位密钥长度，注意初始化时显式调用如下代码：

```
kg.init(168);
```

DESede算法密钥生成器的密钥长度初始化默认值为168位，若使用无参初始化方法，一样会产生一个168位的DESede算法密钥。

如果要使用128位或192位长度的密钥，需要在实例化密钥生成器对象时就指定Bouncy Castle作为该算法的提供者。我们以初始化192位长度密钥为例，按如下代码实现：

```
KeyGenerator kg = KeyGenerator.getInstance(KEY_ALGORITHM, "BC");
kg.init(192);
```

上述密钥长度初始化实现与DES算法实现无差别，这里要注意的是填充方式扩展。

注意代码中的变量“CIPHER_ALGORITHM”，当前指定的填充方式是PKCS5Padding，若我们使用PKCS7Padding填充方式除了对该变量做调整外，还需要调整Cipher对象cipher实例化代码，按如下方式实现：

```
Cipher.getInstance(CIPHER_ALGORITHM, "BC");
```

这样我们就能获得相应填充方式下的加密/解密实现了。

关于DESede算法相应的测试用例与DES算法的测试用例基本没有差别，如代码清单7-4所示。

代码清单7-4 DESede算法实现测试用例

```
import static org.junit.Assert.*;
import org.apache.commons.codec.binary.Base64;
import org.junit.Test;
/**
 * DESede安全编码组件校验
 * @author 梁栋
 * @version 1.0
 */
public class DESedeCoderTest {
    /**
     * 测试
     * @throws Exception
     */
    @Test
    public final void test() throws Exception {
            String inputStr = "DESede";
            byte[] inputData = inputStr.getBytes();
            System.err.println("原文:\t" + inputStr);
```

更多资源请访问稀酷客(www.ckook.com)

```
            // 初始化密钥
            byte[] key = DESedeCoder.initKey();
            System.err.println("密钥:\t" + Base64.encodeBase64String(key));
            // 加密
            inputData = DESedeCoder.encrypt(inputData, key);
            System.err.println("加密后:\t" + Base64.encodeBase64String(inputData));
            // 解密
            byte[] outputData = DESedeCoder.decrypt(inputData, key);
            String outputStr = new String(outputData);
            System.err.println("解密后:\t" + outputStr);
            // 校验
            assertEquals(inputStr, outputStr);
    }
}
```

我们可以从控制台中获得信息中明显感受到密钥长度的增加。控制台输出信息如下所示：

```
原文:       DESede
密钥:       N8jTp6RuZxkjJea2XVvquV5YegHQ31cV
加密后:     touXuJw8vrc=
解密后:     DESede
```

仔细研究过Java API的读者朋友也许会对上述代码中的密钥材料实现类的变化敏感一些，Java API中仅仅提供了DES、DESede和PBE共3种对称加密算法密钥材料实现类。那么，其他算法如何还原密钥呢？AES算法实现就是一个不错的示例！

7.4 高级数据加密标准——AES

DES算法漏洞的发现加速了对称加密算法的改进，通过对DES算法的简单改造得到的DESede算法虽然在一定程度上提升了算法安全强度。但DESede算法低效的加密实现和较慢的处理速度仍不能满足我们对安全的要求。AES算法正是基于这些缘由而诞生。

7.4.1 简述

1997年，NIST发起了征集DES替代算法——AES（Advanced Encryption Standard，高级数据加密标准）算法的活动。1997年9月12日，NIST发布了征集算法的正式公告和具体细节，要求AES算法要比DESede算法快，至少与DESede算法一样安全，具有128位的分组长度，支持128位、192位和256位的密钥长度，同时要求AES要能够在世界范围内免费使用。

1998年8月20日，NIST在“第一次AES候选大会”上公布了满足条件的15个AES的候选算法，继而又从中筛选出5个候选算法，包括MARS、RC6、Rijndael、Serpent和Twofish。

2000年10月2日，由Daemen和Rijmen两位比利时人提出的Rijndael算法，以其密钥设置快、存储要求低，在硬件实现和限制存储的条件下性能优异当选AES算法。

经过验证，目前采用的AES算法能够有效抵御已知的针对DES算法的所有攻击方法，如部

更多资源请访问稀酷客(www.ckook.com)

分差分攻击、相关密钥攻击等。至今，还没有AES破译的官方报道。

AES算法因密钥建立时间短、灵敏性好、内存需求低等优点，在各个领域得到广泛的研究与应用。

目前，AES常用于UMTS（Universal Mobile Telecommunications System，通用移动通信系统）。基于SSH（Secure Shell，安全外壳）协议的一些软件也使用了AES算法。图7-3展示了SecureCRT软件如何配置加密算法。

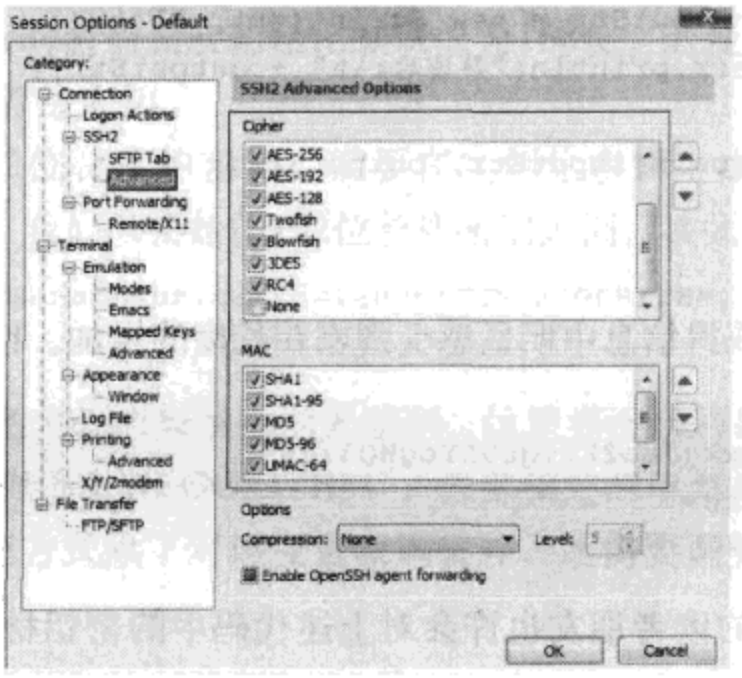


图7-3 SecureCRT配置界面中的加密算法

除了常用的SSH软件外，在一些无线路由器中也使用AES算法构建加密协议。

7.4.2 实现

AES算法成为DES算法的替代者，其实现也成为其他对称加密算法实现的参考模型。无论是Java 6所支持的对称加密算法（如RC2、RC4和Blowfish等算法），或是通过第三方加密组件提供的对称加密算法（如Bouncy Castle提供的IDEA算法实现），都可以通过对AES算法实现做少许改动完成相应实现。本文以AES算法实现为例，读者朋友也可参照实现其他算法，如RC2、RC4和Blowfish等算法。

有关AES算法的Java 6实现细节如表7-3所示。

表7-3 AES算法

算法	密钥长度	密钥长度默认值	工作模式	填充方式	备注
AES	128、192、256	128	ECB、CBC、PCBC、CTR、CTS、CFB、CFB8至CFB128、OFB、OFB8至OFB128	NoPadding、PKCS5Padding、ISO10126Padding	Java 6实现若使用256位密钥需要获得无政策限制权限文件（Unlimited Strength Jurisdiction Policy Files）
	同上	同上	同上	PKCS7Padding、ZeroBytePadding	Bouncy Castle实现

更多资源请访问稀酷客(www.ckook.com)

关于无政策限制权限文件（Unlimited Strength Jurisdiction Policy Files），请参阅第4章相关内容。

AES算法实现基于DES算法实现做了修改，具体实现如代码清单7-5所示。

代码清单7-5 AES算法实现

```
import java.security.Key;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
/**
 * AES安全编码组件
 * @author 梁栋
 * @version 1.0
 */
public abstract class AESCoder {
    /**
     * 密钥算法
     */
    public static final String KEY_ALGORITHM = "AES";
    /**
     * 加密/解密算法 / 工作模式 / 填充方式
     * Java 6支持PKCS5PADDING填充方式
     * Bouncy Castle支持PKCS7Padding填充方式
     */
    public static final String CIPHER_ALGORITHM = "AES/ECB/PKCS5Padding";
    /**
     * 转换密钥
     * @param key 二进制密钥
     * @return Key 密钥
     * @throws Exception
     */
    private static Key toKey(byte[] key) throws Exception {
        // 实例化DES密钥材料
        SecretKey secretKey = new SecretKeySpec(key, KEY_ALGORITHM);
        return secretKey;
    }
    /**
     * 解密
     * @param data 待解密数据
     * @param key 密钥
     * @return byte[] 解密数据
     * @throws Exception
     */
    public static byte[] decrypt(byte[] data, byte[] key) throws Exception {
        // 还原密钥
        Key k = toKey(key);
```

更多资源请访问稀酷客(www.ckook.com)

```
        /*
         * 实例化
         * 使用PKCS7Padding填充方式，按如下方式实现
         * Cipher.getInstance(CIPHER_ALGORITHM, "BC");
         */
        Cipher cipher = Cipher.getInstance(CIPHER_ALGORITHM);
        // 初始化，设置为解密模式
        cipher.init(Cipher.DECRYPT_MODE, k);
        // 执行操作
        return cipher.doFinal(data);
    }
    /**
     * 加密
     * @param data 待加密数据
     * @param key 密钥
     * @return byte[] 加密数据
     * @throws Exception
     */
    public static byte[] encrypt(byte[] data, byte[] key) throws Exception {
        // 还原密钥
        Key k = toKey(key);
        /*
         * 实例化
         * 使用PKCS7Padding填充方式，按如下方式实现
         * Cipher.getInstance(CIPHER_ALGORITHM, "BC");
         */
        Cipher cipher = Cipher.getInstance(CIPHER_ALGORITHM);
        // 初始化，设置为加密模式
        cipher.init(Cipher.ENCRYPT_MODE, k);
        // 执行操作
        return cipher.doFinal(data);
    }
    /**
     * 生成密钥 <br>
     * @return byte[] 二进制密钥
     * @throws Exception
     */
    public static byte[] initKey() throws Exception {
        // 实例化
        KeyGenerator kg = KeyGenerator.getInstance(KEY_ALGORITHM);
        // AES 要求密钥长度为128位、192位或256位
        kg.init(256);
        // 生成秘密密钥
        SecretKey secretKey = kg.generateKey();
        // 获得密钥的二进制编码形式
        return secretKey.getEncoded();
    }
}
```

更多资源请访问稀酷客(www.ckook.com)

上述代码实现当通用的，不仅RC2、RC4和Blowfish算法可以使用上述代码方式实现，包括已经实现的DES和DESede算法都可以参照上述代码实现方式，无须考虑密钥材料实现类，只要对算法名称稍作调整即可！

上述代码测试用例与上文提到的DES和DESede算法的测试用例代码差别不大，如代码清单7-6所示。

代码清单7-6 AES算法实现测试用例

```
import static org.junit.Assert.*;
import org.apache.commons.codec.binary.Base64;
import org.junit.Test;
/**
 * AES安全编码组件校验
 * @author 梁栋
 * @version 1.0
 */
public class AESCoderTest {
    /**
     * 测试
     * @throws Exception
     */
    @Test
    public final void test() throws Exception {
            String inputStr = "AES";
            byte[] inputData = inputStr.getBytes();
            System.err.println("原文:\t" + inputStr);
            // 初始化密钥
            byte[] key = AESCoder.initKey();
            System.err.println("密钥:\t" + Base64.encodeBase64String(key));
            // 加密
            inputData = AESCoder.encrypt(inputData, key);
            System.err.println("加密后:\t" + Base64.encodeBase64String(inputData));
            // 解密
            byte[] outputData = AESCoder.decrypt(inputData, key);
            String outputStr = new String(outputData);
            System.err.println("解密后:\t" + outputStr);
            // 校验
            assertEquals(inputStr, outputStr);
    }
}
```

观察控制台输出的信息，如下所示：

```
原文:       AES
密钥:       4qrBlCmeHHyEAyoNRY2djo1HWx8LLCH2NQHG9c0ahi4=
加密后:     XhWaN6Am1T3NVSFYs1OMVg==
解密后:     AES
```

读者朋友可以以AES算法实现为参考，完成RC2、RC4和Blowfish等算法的实现。

更多资源请访问稀酷客(www.ckook.com)

7.5 国际数据加密标准——IDEA

早在NIST发布征集AES算法以前，就已经有人在找寻DES算法的替代算法了。IDEA算法的提出者未像DESede算法那样在原有DES算法的基础上做改进，而是独辟蹊径地寻求了突破性解决方案。IDEA算法早于AES算法作为DES算法的可选替代算法出现。

7.5.1 简述

IDEA（International Data Encryption Algorithm，国际数据加密标准）算法是由旅居瑞士的中国青年学者来学嘉和著名密码专家James Massey于1990年提出的一种对称分组密码，并于1992年修改完成。

IDEA算法使用长度为128位的密钥，数据块大小为64位。从理论上讲，IDEA属于“强”加密算法，至今还没有出现对该算法的有效攻击算法（以目前计算机水平，破译一个IDEA密钥至少需要10^{13}年）。

IDEA算法在美国之外提出并发展起来，避开了美国法律上对加密技术的诸多限制。因此，有关IDEA算法和实现技术的书籍均可自由出版和交流，极大地促进了IDEA算法的发展和完善。

IDEA算法是目前较为常用的电子邮件加密算法之一。电子邮件加密软件PGP（Pretty Good Privacy）使用了具有商业版权的IDEA算法，实现邮件加密/解密工作。

7.5.2 实现

Java 6没有提供IDEA算法的相应实现，若需要使用该算法我们可以通过Bouncy Castle来完成。Bouncy Castle不仅提供了IDEA算法实现，包括其他Java 6未能支持的对称加密算法也可通过Bouncy Castele实现，如AES候选算法Rijndael、Serpent和Twofish等。本文以IDEA算法实现为例，读者朋友可参照其他算法的实现，如Rijndael、Serpent和Twofish等。

有关IDEA算法的Bouncy Castle实现细节如表7-4所示。

表7-4 IDEA算法

算法	密钥长度	密钥长度默认值	工作模式	填充方式	备注
IDEA	128	128	ECB	PKCS5Padding、PKCS7Padding、ISO10126Padding、ZeroBytePadding	Bouncy Castle实现

IDEA算法的实现需要通过Bouncy Castle来完成，参考如下代码：

```
import java.security.Security;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
// 省略
// 加入BouncyCastleProvider支持
Security.addProvider(new BouncyCastleProvider());
```

更多资源请访问稀酷客(www.ckook.com)

本节代码实现对于其他Bouncy Castle支持的算法实现有代表性意义，实现如代码清单7-7所示。

代码清单7-7 IDEA算法实现

```
import java.security.Key;
import java.security.Security;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
/**
 * IDEA安全编码组件
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class IDEACoder {
    /**
     * 密钥算法
     */
    public static final String KEY_ALGORITHM = "IDEA";
    /**
     * 加密/解密算法 / 工作模式 / 填充方式
     */
    public static final String CIPHER_ALGORITHM = "IDEA/ECB/ISO10126Padding";
    /**
     * 转换密钥
     * @param key 二进制密钥
     * @return Key 密钥
     * @throws Exception
     */
    private static Key toKey(byte[] key) throws Exception {
        // 生成秘密密钥
        SecretKey secretKey = new SecretKeySpec(key, KEY_ALGORITHM);
        return secretKey;
    }
    /**
     * 解密
     * @param data 待解密数据
     * @param key 密钥
     * @return byte[] 解密数据
     * @throws Exception
     */
    public static byte[] decrypt(byte[] data, byte[] key) throws Exception {
        // 加入BouncyCastleProvider支持
        Security.addProvider(new BouncyCastleProvider());
```

更多资源请访问稀酷客(www.ckook.com)

```
        // 还原密钥
        Key k = toKey(key);
        // 实例化
        Cipher cipher = Cipher.getInstance(CIPHER_ALGORITHM);
        // 初始化，设置为解密模式
        cipher.init(Cipher.DECRYPT_MODE, k);
        // 执行操作
        return cipher.doFinal(data);
    }
    /**
     * 加密
     * @param data 待加密数据
     * @param key 密钥
     * @return byte[] 加密数据
     * @throws Exception
     */
    public static byte[] encrypt(byte[] data, byte[] key) throws Exception {
        // 加入BouncyCastleProvider支持
        Security.addProvider(new BouncyCastleProvider());
        // 还原密钥
        Key k = toKey(key);
        // 实例化
        Cipher cipher = Cipher.getInstance(CIPHER_ALGORITHM);
        // 初始化，设置为加密模式
        cipher.init(Cipher.ENCRYPT_MODE, k);
        // 执行操作
        return cipher.doFinal(data);
    }
    /**
     * 生成密钥 <br>
     * @return byte[] 二进制密钥
     * @throws Exception
     */
    public static byte[] initKey() throws Exception {
        // 加入BouncyCastleProvider支持
        Security.addProvider(new BouncyCastleProvider());
        // 实例化
        KeyGenerator kg = KeyGenerator.getInstance(KEY_ALGORITHM);
        // 初始化
        kg.init(128);
        // 生成秘密密钥
        SecretKey secretKey = kg.generateKey();
        // 获得密钥的二进制编码形式
        return secretKey.getEncoded();
    }
}
```

IDEA算法本身也可以通过Bouncy Castle的API来实现，甚至包括其他Java 6未能支持的算

更多资源请访问稀酷客(www.ckook.com)

法也可以通过Bouncy Castle的API来实现，包括Serpent和Twofish等。但是，如果这样做就跳出了JCE框架，增加了学习难度。既然JCE框架给我们提供了便捷灵活的第三方加密算法调用方式，我们为什么不能加以运用呢？

对于上述代码的测试如代码清单7-8所示。

代码清单7-8 IDEA算法实现测试用例

```
import static org.junit.Assert.*;
import org.apache.commons.codec.binary.Base64;
import org.junit.Test;
/**
 * IDEA安全编码组件校验
 * @author 梁栋
 * @version 1.0
 */
public class IDEACoderTest {
    /**
     * 测试
     * @throws Exception
     */
    @Test
    public final void test() throws Exception {
            String inputStr = "IDEA";
            byte[] inputData = inputStr.getBytes();
            System.err.println("原文:\t" + inputStr);
            // 初始化密钥
            byte[] key = IDEACoder.initKey();
            System.err.println("密钥:\t" + Base64.encodeBase64String(key));
            // 加密
            inputData = IDEACoder.encrypt(inputData, key);
            System.err.println("加密后:\t" + Base64.encodeBase64String(inputData));
            // 解密
            byte[] outputData = IDEACoder.decrypt(inputData, key);
            String outputStr = new String(outputData);
            System.err.println("解密后:\t" + outputStr);
            // 校验
            assertEquals(inputStr, outputStr);
    }
}
```

控制台输出信息如下所示：

```
原文:       IDEA
密钥:       W1a1IFCKIbXKdnejfT5b2A==
加密后:     er43StU2Txk=
解密后:     IDEA
```

更多资源请访问稀酷客(www.ckook.com)

7.6 基于口令加密——PBE

前文阐述了几种常用的对称加密算法，这些算法的应用模型（甚至包括实现）几乎同出一辙。但并不是所有的对称加密算法都是如此，PBE算法综合了上述对称加密算法和消息摘要算法的优势，形成了对称加密算法的一个特例。

7.6.1 简述

PBE（Password Based Encryption，基于口令加密）算法是一种基于口令的加密算法，其特点在于口令由用户自己掌管，采用随机数（这里我们叫做盐）杂凑多重加密等方法保证数据的安全性。

PBE算法没有密钥的概念，密钥在其他对称加密算法中是经过算法计算得出的，PBE算法中则使用口令替代了密钥。

密钥的长短直接影响了算法的安全性，但不方便记忆。即便是我们将密钥通过Base64编码转换为可见字符，长密钥一样不容易记忆。因此，在这种情况下密钥是需要存储的，但是口令则不然。作者天天都要开关电脑，进入操作系统的唯一途径就是输入口令。口令是我们便于记忆的一种凭证，基于这一点，PBE算法使用口令代替了密钥。

PBE算法并没有真正构建新的加密/解密算法，而是对我们已知的对称加密算法（如DES算法）做了包装。使用PBE算法对数据做加密/解密操作时，其实是使用了DES或AES等其他对称加密算法做了相应的操作。

既然PBE算法使用了我们较为常用的对称加密算法，那就无法回避密钥的问题。口令并不能替代密钥，密钥是经过加密算法计算得出的，但口令本身不可能很长，单纯的口令很容易通过穷举攻击方式破译，这就引入了“盐”。盐能够阻止字典攻击或预先计算的攻击，它本身是一个随机信息，相同的随机信息极不可能使用两次。将盐附加在口令上，通过消息摘要算法经过迭代计算获得构建密钥/初始化向量的基本材料，使得破译加密信息的难度加大。

PBE算法是对称加密算法的综合性算法，常见算法如PBEWithMD5AndDES，该算法使用了MD5和DES算法构建PBE算法。

7.6.2 模型分析

基于PBE算法的消息传递模型与基于DES算法的消息传递模型还是有一定差别的，如图7-4所示。

甲乙双方作为消息传递双方（甲方作为发送方，乙方作为接收方），我们假定甲乙双方在消息传递前已商定加密算法和迭代次数，欲完成一次消息传递需经过如下步骤：

1）由消息传递双方约定口令，这里由甲方构建口令。

2）由口令构建者公布口令，这里由甲方将口令公布给乙方。

3）由口令构建者构建本次消息传的使用的盐，这里由甲方构建盐。

4）由消息发送方使用口令、盐对数据加密，这里由甲方对数据加密。

更多资源请访问稀酷客(www.ckook.com)

5）由消息发送方将盐、加密数据发送给消息接收者，这里由甲方将盐、加密数据发送给乙方。

6）由消息接收方使用盐、口令对加密数据解密，这里由乙方完成数据解密。

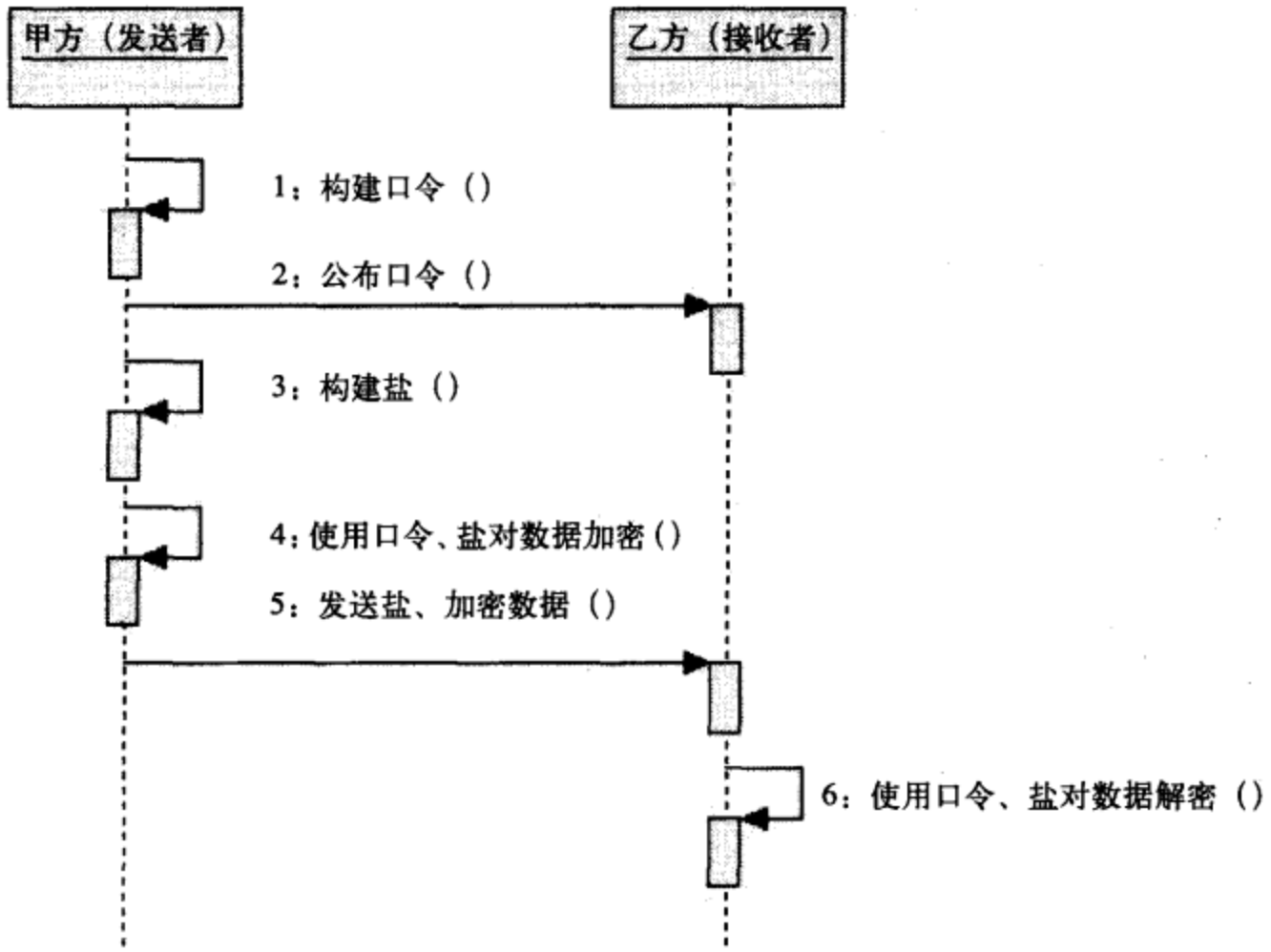


图7-4 基于PBE算法的消息通信模型

基于PBE算法的消息传递模型理解起来并不十分复杂。对于上述单向消息传递而言，如果乙方想要回复甲方消息，甲乙并不需要重复步骤1、2，仅仅需由乙方执行步骤3、4、5，由甲方执行步骤6即可。

当然，甲乙双方也可以在消息传递过程中传递迭代次数。

我们可以设想，“盐”本身是一种可以由消息传递双方按一定规律约定的信息，譬如时间。也可以是某个不可变物理硬件的编号，譬如U盘的自身唯一标识。

甲乙双方可以通过约定消息传递时间，并将其作为“盐”的基本信息，根据预定算法（如MD5算法）对其处理，最终获得真正的“盐”。这样一来，“盐”就无需传递，提高了安全性。

换一种思路考虑，有这样一种系统，使用者需要将U盘插入计算机，同时输入口令方能登录系统，那么U盘就是“盐”信息的提供者！即使U盘丢失，加密的信息也未必能被窃取！

从另一个角度思考，“盐”与口令就像两把不可分割的钥匙。

7.6.3 实现

Java 6和Bouncy Castle都提供了PBE系列算法的相关实现，差别在于对各种消息摘要算法和对称加密算法的组合。常用的消息摘要算法包括MD5和SHA算法，常用的对称加密算法包括DES、RC2等。PBE系列算法就是将这些算法进行合理组合，其密钥长度均以PBE具体算法中

更多资源请访问稀酷客(www.ckook.com)

的对称加密算法密钥长度为准，其工作模式基本上为CBC模式。

有关PBE算法的Java 6和Bouncy Castle实现细节如表7-5所示。

表7-5 PBE算法

算法	密钥长度	密钥长度默认值	工作模式	填充方式	备注
PBEWithMD5AndDES	56	56	CBC	PKCS5Padding	Java 6实现
PBEWithMD5AndTripleDES	112、168	168			
PBEWithSHA1AndDESede	112、168	168			
PBEWithSHA1AndRC2_40	40至1024（8的倍数）	128			
PBEWithMD5AndDES	64	64	CBC	PKCS5Padding、PKCS7Padding、ISO10126Padding、ZeroBytePadding	Bouncy Castle实现
PBEWithMD5AndRC2	128	128			
PBEWithSHA1AndDES	64	64			
PBEWithSHA1AndRC2	128	128			
PBEWithSHAAndIDEA-CBC	128	128			
PBEWithSHAAnd2-KeyTripleDES-CBC	128	128			
PBEWithSHAAnd3-KeyTripleDES-CBC	192	192			
PBEWithSHAAnd128BitRC2-CBC	128	128			
PBEWithSHAAnd40BitRC2-CBC	40	40			
PBEWithSHAAnd128BitRC4	128	128			
PBEWithSHAAnd40BitRC4	40	40			
PBEWithSHAAndTwofish-CBC	256	256			

PEB算法是实现过程中需要关注的环节，包括盐的初始化、密钥材料的转换以及加密/解密实现。

在初始化盐时，必须使用随机的方式构建盐，最终要得到一个8字节的字节数组。鉴于安全性要求，这里的随机数生成器只能使用SecureRandom类，如下所示：

```
// 实例化安全随机数
SecureRandom random = new SecureRandom();
// 产出盐
byte[] b = random.generateSeed(8);
```

字节数组b[]就是我们要的盐。

密钥材料转换部分不同于其他对称加密算法，这里使用的是PBEKeySpec类，如下所示：

```
// 密钥材料转换
PBEKeySpec keySpec = new PBEKeySpec(password.toCharArray());
```

其他对称加密算法的密钥材料实现类的构造方法要求输入字节数组形式的变量，而PBEKeySpec类构造方法则要求输入字符数组变量。

为什么不是字符串（String）而是字符数组（char[]）呢？这是因为字符串是可序列化的封装类，可在程序调用时被序列化到文件中，而字符数组只能以内存变量的形式保留在内存中。

在加密/解密实现时，需要注意使用参数材料PBEParameterSpec类，同时注意迭代次数。构

更多资源请访问稀酷客(www.ckook.com)

建PBE参数材料后就可以转交给Cipher类完成加密/解密操作，如下所示：

```
// 实例化PBE参数材料
PBEParameterSpec paramSpec = new PBEParameterSpec(salt, ITERATION_COUNT);
// 实例化
Cipher cipher = Cipher.getInstance(ALGORITHM);
// 初始化
cipher.init(Cipher.DECRYPT_MODE, key, paramSpec);
```

本文以Java 6提供的PBEWITHMD5andDES算法为例，如代码清单7-9所示。

代码清单7-9 PBE算法实现

```
import java.security.Key;
import java.security.SecureRandom;
import javax.crypto.Cipher;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;
import javax.crypto.spec.PBEParameterSpec;
/**
 * PBE安全编码组件
 * @author 梁栋
 * @version 1.0
 */
public abstract class PBECoder {
    /**
     * Java 6 支持以下任意一种算法
     * PBEWithMD5AndDES
     * PBEWithMD5AndTripleDES
     * PBEWithSHA1AndDESede
     * PBEWithSHA1AndRC2_40
     * PBKDF2WithHmacSHA1
     */
    public static final String ALGORITHM = "PBEWITHMD5andDES";
    /**
     * 迭代次数
     */
    public static final int ITERATION_COUNT = 100;
    /**
     * 盐初始化<br>
     * 盐长度必须为8字节
     * @return byte[] 盐
     * @throws Exception
     */
    public static byte[] initSalt() throws Exception {
        // 实例化安全随机数
        SecureRandom random = new SecureRandom();
        // 产出盐
```

更多资源请访问稀酷客(www.ckook.com)

```
        return random.generateSeed(8);
    }
    /**
     * 转换密钥
     * @param password 密码
     * @return Key 密钥
     * @throws Exception
     */
    private static Key toKey(String password) throws Exception {
        // 密钥材料转换
        PBEKeySpec keySpec = new PBEKeySpec(password.toCharArray());
        // 实例化
        SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(ALGORITHM);
        // 生成密钥
        SecretKey secretKey = keyFactory.generateSecret(keySpec);
        return secretKey;
    }
    /**
     * 加密
     * @param data 数据
     * @param password 密码
     * @param salt 盐
     * @return byte[] 加密数据
     * @throws Exception
     */
    public static byte[] encrypt(byte[] data, String password, byte[] salt)
    throws Exception {
        // 转换密钥
        Key key = toKey(password);
        // 实例化PBE参数材料
        PBEParameterSpec paramSpec = new PBEParameterSpec(salt, ITERATION_COUNT);
        // 实例化
        Cipher cipher = Cipher.getInstance(ALGORITHM);
        // 初始化
        cipher.init(Cipher.ENCRYPT_MODE, key, paramSpec);
        // 执行操作
        return cipher.doFinal(data);
    }
    /**
     * 解密
     * @param data 数据
     * @param password 密码
     * @param salt 盐
     * @return byte[] 解密数据
     * @throws Exception
     */
    public static byte[] decrypt(byte[] data, String password, byte[] salt)
```

更多资源请访问稀酷客(www.ckook.com)

```
    throws Exception {
            // 转换密钥
            Key key = toKey(password);
            // 实例化PBE参数材料
            PBEParameterSpec paramSpec = new PBEParameterSpec(salt,
            ITERATION_COUNT);
            // 实例化
            Cipher cipher = Cipher.getInstance(ALGORITHM);
            // 初始化
            cipher.init(Cipher.DECRYPT_MODE, key, paramSpec);
            // 执行操作
            return cipher.doFinal(data);
    }
}
```

上述代码的测试用例如代码清单7-10所示。

代码清单7-10 PBE算法实现测试用例

```
import static org.junit.Assert.*;
import org.apache.commons.codec.binary.Base64;
import org.junit.Test;
/**
 * PBE校验
 * @author 梁栋
 * @version 1.0
 */
public class PBECoderTest {
    /**
     * 测试
     * @throws Exception
     */
    @Test
    public void test() throws Exception {
            String inputStr = "PBE";
            System.err.println("原文: " + inputStr);
            byte[] input = inputStr.getBytes();
            String pwd = "snowolf@zlex.org";
            System.err.println("密码: \t" + pwd);
            // 初始化盐
            byte[] salt = PBECoder.initSalt();
            System.err.println("盐: \t" + Base64.encodeBase64String(salt));
            // 加密
            byte[] data = PBECoder.encrypt(input, pwd, salt);
            System.err.println("加密后: \t" + Base64.encodeBase64String(data));
            // 解密
            byte[] output = PBECoder.decrypt(data, pwd, salt);
```

更多资源请访问稀酷客(www.ckook.com)

```
            String outputStr = new String(output);
            System.err.println("解密后: \t" + outputStr);
            // 校验
            assertEquals(inputStr, outputStr);
    }
}
```

我们在控制台中得到如下信息：

```
原文:       PBE
密码:       snowolf@zlex.org
盐:         qbjve/LfGIM=
加密后:     NZQG0WfqAg4=
解密后:     PBE
```

7.7 实例：对称加密网络应用

加密技术与网络应用密不可分，在开放的网络中进行机密数据传输少不了使用加密算法。在实际应用中，我们常常需要向合作伙伴发送XML格式的数据包。例如，数据报表、货仓清单等。这些都是机密信息，如不对其加密必将公司机密泄露无疑。

本文将使用AES算法，并配合第6章提到的SHA算法构建简单的基于对称加密算法的数据传输网络应用——DataServer。

本实例将用到开源组件包Commons Codec来协助完成本实例的构建。请读者朋友将该相关jar包部署到该应用的/WEB-INF/lib目录下。

首先，我们来构建一个用于HTTP请求的工具类——HttpUtils。当然，读者朋友可以选用成熟的开源框架——Apache Commons HttpClient替代下述代码。相信大多数读者朋友对于如何构建HTTP请求都了如指掌，为避免拖沓作者在此不对该类做详细介绍，请读者朋友参考Java API相关内容。代码实现如代码清单7-11所示。

代码清单7-11 HttpUtils

```
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.Map;
import java.util.Properties;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
/**
 * Http 工具
```

更多资源请访问稀酷客(www.ckook.com)

```
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class HttpUtils {
    public static final String CHARACTER_ENCODING = "UTF-8";
    public static final String METHOD_POST = "POST";
    public static final String CONTENT_TYPE = "Content-Type";
    /**
     * 打印数据
     * @param response HttpServletResponse
     * @param data 待打印的数据
     */
    public static void responseWrite(HttpServletResponse response, byte[] data)
    throws IOException {
        if (data != null) {
            response.setContentLength(data.length);
            DataOutputStream out = new DataOutputStream(response
                    .getOutputStream());
            out.write(data);
            out.flush();
            out.close();
        }
    }
    /**
     * 从请求中读字节流
     * @param request HttpServletRequest
     * @return byte[] 数据
     * @throws IOException
     */
    public static byte[] requestRead(HttpServletRequest request)
            throws IOException {
        int contentLength = request.getContentLength();
        byte[] data = null;
        if (contentLength > 0) {
            data = new byte[contentLength];
            InputStream is = request.getInputStream();
            DataInputStream dis = new DataInputStream(is);
            dis.readFully(data);
            dis.close();
        }
        return data;
    }
    /**
     * 以POST方式向指定地址发送数据包请求,并取得返回的数据包
     * @param urlString 请求地址
     * @param requestData 请求数据
     * @return byte[] 数据包
```

更多资源请访问稀酷客(www.ckook.com)

```
     * @throws IOException
     */
    public static byte[] postRequest(String urlString, byte[] requestData)
    throws Exception {
        Properties requestProperties = new Properties();
        requestProperties.setProperty(CONTENT_TYPE, "application/octet-stream;
        charset=" + CHARACTER_ENCODING);
        return postRequest(urlString, requestData, requestProperties);
    }
    /**
     * 以POST方式向指定地址发送数据包请求,并取得返回的数据包
     * @param urlString 请求地址
     * @param requestData 请求数据
     * @param requestProperties 请求包体
     * @return byte[] 数据包
     * @throws IOException
     */
    public static byte[] postRequest(String urlString, byte[] requestData,
    Properties requestProperties) throws Exception {
        byte[] responseData = null;
        HttpURLConnection con = null;
        try {
            URL url = new URL(urlString);
            con = (HttpURLConnection) url.openConnection();
            if ((requestProperties != null) && (requestProperties.size() > 0)) {
                for (Map.Entry<Object, Object> entry : requestProperties.entrySet()) {
                    String key = String.valueOf(entry.getKey());
                    String value = String.valueOf(entry.getValue());
                    con.setRequestProperty(key, value);
                }
            }
            con.setRequestMethod(METHOD_POST);
            con.setDoOutput(true);
            con.setDoInput(true);
            DataOutputStream dos = new DataOutputStream(con.getOutputStream());
            if (requestData != null) {
                dos.write(requestData);
            }
            dos.flush();
            dos.close();
            DataInputStream dis = new DataInputStream(con.getInputStream());
            int length = con.getContentLength();
            if (length > 0) {
                responseData = new byte[length];
                dis.readFully(responseData);
            }
            dis.close();
        } finally {
```

更多资源请访问稀酷客(www.ckook.com)

```
            // 关闭流
            if (con != null) {
                con.disconnect();
                con = null;
            }
        }
        return responseData;
    }
}
```

接下来，我们将构建用于加密/解密的工具类——AESCoder。此处，我们将用到第三方开源组件包——Commons Codec。

对于AESCoder类，我们参考代码清单7-5稍作调整，对密钥稍作包装以方便在存储和使用，如代码清单7-12所示。

代码清单7-12 AESCoder——密钥封装

```
import org.apache.commons.codec.binary.Base64;
// 省略
/**
 * 初始化密钥
 * @return String Base64编码密钥
 * @throws Exception
 */
public static String initKeyString() throws Exception {
    return Base64.encodeBase64String(initKey());
}
/**
 * 获取密钥
 * @param key 密钥
 * @return byte[] 密钥
 * @throws Exception
 */
public static byte[] getKey(String key) throws Exception {
    return Base64.decodeBase64(key);
}
/**
 * 解密
 * @param data 待解密数据
 * @param key 密钥
 * @return byte[] 解密数据
 * @throws Exception
 */
public static byte[] decrypt(byte[] data, String key) throws Exception {
    return decrypt(data, getKey(key));
}
/**
 * 加密
```

更多资源请访问稀酷客(www.ckook.com)

```
     * @param data 待加密数据
     * @param key 密钥
     * @return byte[] 加密数据
     * @throws Exception
     */
    public static byte[] encrypt(byte[] data, String key) throws Exception {
        return encrypt(data, getKey(key));
    }
```

这里我们使用Commons Codec提供Base64算法对密钥进行封装/解包。通过initKeyString()方法我们将得到密钥字符串："Zk6tc8Gg3NVi+m6X2UmV7rd1XRRiF1sUNtxFbiFqjMs="。我们将在后续操作中使用到该内容。

为防止传递的机密数据在网络传递过程中被篡改，我们可以对数据进行消息摘要，并对该摘要进行验证。这里使用SHA算法对数据进行摘要/验证。完整实现如代码清单7-13所示。

代码清单7-13 摘要/验证

```
import org.apache.commons.codec.digest.DigestUtils;
// 省略
/**
 * 摘要处理
 * @param data 待摘要数据
 * @return String 摘要字符串
 */
public static String shaHex(byte[] data) {
    return DigestUtils.md5Hex(data);
}
/**
 * 验证
 * @param data 待摘要数据
 * @param messageDigest 摘要字符串
 * @return boolean 验证结果
 */
public static boolean validate(byte[] data, String messageDigest) {
    return messageDigest.equals(shaHex(data));
}
```

AESCoder类完整实现如代码清单7-14所示。

代码清单7-14 AESCoder

```
import java.security.Key;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import org.apache.commons.codec.binary.Base64;
import org.apache.commons.codec.digest.DigestUtils;
/**
```

更多资源请访问稀酷客(www.ckook.com)

```
 * AES安全编码组件
 * @author 梁栋
 * @version 1.0
 */
public abstract class AESCoder {
    /**
     * 密钥算法
     */
    public static final String ALGORITHM = "AES";
    /**
     * 转换密钥
     * @param key 二进制密钥
     * @return Key 密钥
     * @throws Exception
     */
    private static Key toKey(byte[] key) throws Exception {
            // 实例化AES密钥材料
            SecretKey secretKey = new SecretKeySpec(key, ALGORITHM);
            return secretKey;
    }
    /**
     * 解密
     * @param data 待解密数据
     * @param key 密钥
     * @return byte[] 解密数据
     * @throws Exception
     */
    public static byte[] decrypt(byte[] data, byte[] key) throws Exception {
            // 还原密钥
            Key k = toKey(key);
            // 实例化
            Cipher cipher = Cipher.getInstance(ALGORITHM);
            // 初始化，设置为解密模式
            cipher.init(Cipher.DECRYPT_MODE, k);
            // 执行操作
            return cipher.doFinal(data);
    }
    /**
     * 解密
     * @param data 待解密数据
     * @param key 密钥
     * @return byte[] 解密数据
     * @throws Exception
     */
    public static byte[] decrypt(byte[] data, String key) throws Exception {
           return decrypt(data, getKey(key));
    }
    /**
```

更多资源请访问稀酷客(www.ckook.com)

```
     * 加密
     * @param data 待加密数据
     * @param key 密钥
     * @return byte[] 加密数据
     * @throws Exception
     */
    public static byte[] encrypt(byte[] data, byte[] key) throws Exception {
            // 还原密钥
            Key k = toKey(key);
            // 实例化
            Cipher cipher = Cipher.getInstance(ALGORITHM);
            // 初始化，设置为加密模式
            cipher.init(Cipher.ENCRYPT_MODE, k);
            // 执行操作
            return cipher.doFinal(data);
    }
    /**
     * 加密
     * @param data 待加密数据
     * @param key 密钥
     * @return byte[] 加密数据
     * @throws Exception
     */
    public static byte[] encrypt(byte[] data, String key) throws Exception {
            return encrypt(data, getKey(key));
    }
    /**
     * 生成密钥
     * @return byte[] 二进制密钥
     * @throws Exception
     */
    public static byte[] initKey() throws Exception {
            // 实例化
            KeyGenerator kg = KeyGenerator.getInstance(ALGORITHM);
            // 初始化256位密钥
            kg.init(256);
            // 生成秘密密钥
            SecretKey secretKey = kg.generateKey();
            // 获得密钥的二进制编码形式
            return secretKey.getEncoded();
    }
    /**
     * 初始化密钥
     * @return String Base64编码密钥
     * @throws Exception
     */
    public static String initKeyString() throws Exception {
            return Base64.encodeBase64String(initKey());
```

更多资源请访问稀酷客(www.ckook.com)

```
    }
    /**
     * 获取密钥
     * @param key 密钥
     * @return byte[] 密钥
     * @throws Exception
     */
    public static byte[] getKey(String key) throws Exception {
            return Base64.decodeBase64(key);
    }
    /**
     * 摘要处理
     * @param data 待摘要数据
     * @return String 摘要字符串
     */
    public static String shaHex(byte[] data) {
            return DigestUtils.md5Hex(data);
    }
    /**
     * 验证
     * @param data 待摘要数据
     * @param messageDigest 摘要字符串
     * @return boolean 验证结果
     */
        public static boolean validate(byte[] data, String messageDigest) {
                return messageDigest.equals(shaHex(data));
        }
    }
```

接下来，我们将构建用于提供服务的Servlet——DataServlet类。这里，DataServlet类继承了HttpServlet类，重写了init()方法并实现了doPost()方法。

我们希望将密钥可以通过web.xml文件进行配置，在DataServlet启动时加载该密钥。我们通过重写init()方法获得密钥信息。相关实现如代码清单7-15所示。

代码清单7-15 DataServlet——初始化

```
// 省略
// 密钥
private static String key;
// Servlet初始化参数—密钥
private static final String KEY_PARAM = "key";
// 省略
// 初始化
@Override
public void init() throws ServletException {
    super.init();
    // 初始化密钥
    key = getInitParameter(KEY_PARAM);
}
```

更多资源请访问稀酷客(www.ckook.com)

我们可以调用HttpUtils类的requestRead()方法获得请求内容，调用AESCoder类的decrypt()方法对请求内容解密，并将解密后的内容输出在控制台中。同时，我们将从HTTP Header中获得此次交互数据的摘要信息，并对此验证。如果验证通过，则回复“OK”。相关实现如代码清单7-16所示。

代码清单7-16 DataServlet——处理POST请求

```
// HTTP Header 摘要参数名
private static final String HEAD_MD = "messageDigest";
// 省略
// 处理POST请求
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    try {
            byte[] input = HttpUtils.requestRead(request);
            // 对数据解密
            byte[] data = AESCoder.decrypt(input, key);
            System.err.println(new String(data));
            // 默认回复内容
            byte[] output = "".getBytes();
            // 获得此次交互数据的摘要信息
            String messageDigest = request.getHeader(HEAD_MD);
            // 如果验证成功则回复OK
            if (AESCoder.validate(data, messageDigest)) {
                // 如果正常接收到数据则回复OK
                output = "OK".getBytes();
            }
            // 加密回复
            HttpUtils.responseWrite(response, AESCoder.encrypt(output, key));
        } catch (Exception e) {
            throw new ServletException(e);
        }
}
```

完整实现如代码清单7-17所示。

代码清单7-17 DataServlet

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
/**
 * 数据服务DataServlet
 * @author 梁栋
 * @since 1.0
 */
```

更多资源请访问稀酷客(www.ckook.com)

```
public class DataServlet extends HttpServlet {
    private static final long serialVersionUID = -6219906900195793155L;
    // 密钥
    private static String key;
    // Servlet初始化参数—密钥
    private static final String KEY_PARAM = "key";
    // HTTP Header 摘要参数名
    private static final String HEAD_MD = "messageDigest";
    // 初始化
    @Override
    public void init() throws ServletException {
            super.init();
            // 初始化密钥
            key = getInitParameter(KEY_PARAM);
    }
    // 处理POST请求
    protected void doPost(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
            try {
                    // 获得此次交互数据的摘要信息
                    String messageDigest = request.getHeader(HEAD_MD);
                    byte[] input = HttpUtils.requestRead(request);
                    // 对数据解密
                    byte[] data = AESCoder.decrypt(input, key);
                    System.err.println(new String(data));
                    // 默认回复内容
                    byte[] output = "".getBytes();
                    // 如果验证成功则回复OK
                    if (AESCoder.validate(data, messageDigest)) {
                    // 如果正常接收到数据则回复OK
                    output = "OK".getBytes();
                }
                // 加密回复
                HttpUtils.responseWrite(response, AESCoder.encrypt(output, key));
            } catch (Exception e) {
                throw new ServletException(e);
            }
        }
    }
```

此处，我们将密钥（“Zk6tc8Gg3NVi+m6X2UmV7rd1XRRiF1sUNtxFbiFqjMs=”）配置在web.xml文件中，完整代码如代码清单7-18所示。

代码清单7-18 web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

更多资源请访问稀酷客(www.ckook.com)

```
        xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
        id="WebApp_ID"
        version="2.5">
        <display-name>DataServer</display-name>
        <servlet>
            <servlet-name>DataServlet</servlet-name>
            <servlet-class>DataServlet</servlet-class>
            <init-param>
                <param-name>key</param-name>
                <param-value><![CDATA[Zk6tc8Gg3NVi+m6X2UmV7rdlXRRiF1sUNtxFbiFqjMs=]]></param-value>
            </init-param>
        </servlet>
        <servlet-mapping>
            <servlet-name>DataServlet</servlet-name>
            <url-pattern>/DataServlet</url-pattern>
        </servlet-mapping>
    </web-app>
```

此处，我们将向服务器发送XML格式的数据包。并在发送数据之前对其进行摘要和加密，并校验获得的信息是否包含“OK”内容。此处，我们将密钥作为变量写入DataServletTest类中。完整实现如代码清单7-19所示。

代码清单7-19 DataServletTest

```
import static org.junit.Assert.*;
import java.util.Properties;
import org.junit.Test;
/**
 * DataServlet测试用例
 * @author 梁栋
 * @since 1.0
 */
public class DataServletTest {
    // 秘密密钥
    private static final String key = "Zk6tc8Gg3NVi+m6X2UmV7rdlXRRiF1sUNtxFbiFqjMs=";
    // 请求地址
    private static final String url = "http://localhost:8080/dataserver/DataServlet";
    @Test
    public final void test() throws Exception {
        // 构造数据包
        StringBuilder sb = new StringBuilder();
        sb.append("<?xml version=\"1.0\" encoding=\"UTF-8\"?>\r\n");
        sb.append("<dataGroup>\r\n");
        sb.append("\t<dataItem>\r\n");
        sb.append("\t\t<id>");
        sb.append("10201");
```

更多资源请访问稀酷客(www.ckook.com)

```
            sb.append("</id>\r\n");
            sb.append("\t\t<price>");
            sb.append("35.0");
            sb.append("</price>\r\n");
            sb.append("\t\t<time>");
            sb.append("2009-10-30");
            sb.append("</time>\r\n");
            sb.append("\t</dataItem>\r\n");
            sb.append("\t<dataItem>\r\n");
            sb.append("\t\t<id>");
            sb.append("10301");
            sb.append("</id>\r\n");
            sb.append("\t\t<price>");
            sb.append("55.0");
            sb.append("</price>\r\n");
            sb.append("\t\t<time>");
            sb.append("2009-10-31");
            sb.append("</time>\r\n");
            sb.append("\t</dataItem>\r\n");
            sb.append("</dataGroup>\r\n");
            byte[] data = sb.toString().getBytes();
            // 向HTTP Header附加数据摘要信息
            Properties requestProperties = new Properties();
            requestProperties.put("messageDigest", AESCoder.shaHex(data));
            // 加密并发送数据
            byte[] input = HttpUtils.postRequest(url, AESCoder.encrypt(data,
            key), requestProperties);
            // 解密
            input = AESCoder.decrypt(input, key);
            // 校验
            assertEquals("OK", new String(input));
        }
    }
```

启动DataServer服务，并执行测试用例，我们将在控制台中得到如下信息：

```
<?xml version="1.0" encoding="UTF-8"?>
<dataGroup>
    <dataItem>
        <id>10201</id>
        <price>35.0</price>
        <time>2009-10-30</time>
    </dataItem>
    <dataItem>
        <id>10301</id>
        <price>55.0</price>
        <time>2009-10-31</time>
    </dataItem>
</dataGroup>
```

更多资源请访问稀酷客(www.ckook.com)

很显然，这是我们发送给服务器的数据内容。

在实际应用中，我们经常需要与合作伙伴进行机密数据交互。通常，我们都会使用对称加密算法，例如AES算法对其进行加密。并使用消息摘要算法对其内容进行摘要/验证，例如SHA算法。

对称加密算法非常适用于一般中小型加密网络应用。进行交互的两方应用可能都是定制的系统，密钥在系统设计当初就已敲定。而对于大中型加密网络应用来讲，这样的密钥管理就比较繁琐了。希望上述简单的代码实现可以为读者朋友构建加密网络应用起到抛砖引玉的作用。

7.8 小结

对称加密算法经由古典加密算法演化而来，是现代密码学的重要组成部分。对于大多数对称加密算法而言，解密算法是加密算法的逆运算，加密密钥和解密密钥相同，这也是对称加密算法名称的由来。

对称加密算法家族庞大，迄今为止约有20多种。比较具有代表性的算法有5种：DES、DESede、AES、IDEA和PBE。

DES（Data Encryption Standard，数据加密标准）算法最具代表性，由IBM最先提出并经美国国家标准局制定为数据加密标准，堪称经典对称加密算法，为后续对称加密算法的发展奠定了坚实的基础。自20世纪70年代至20世纪末，DES算法一直稳坐对称加密算法宝座。历经20年发展，DES算法不仅应用在软件行业成为电子商务必不可少的加密算法，同时也逐步渗透到硬件行业。

Java 6支持DES算法实现，支持56位密钥长度；通过Bouncy Castle可将DES算法密钥延长至64位。

越来越多的密码学研究机构发现，密钥长度仅有56位的DES算法越来越不安全。DES算法有三点安全隐患：密钥太短、迭代偏少和半公开性。这使得DES算法的淘汰成为一种必然，但要淘汰DES算法必须找到合适的替代方案。

针对密钥太短和迭代偏少的问题，有人提出了多重DES的方式来克服这些缺陷。比较典型的有两重DES（2DES）、三重DES（3DES）和四重DES（4DES）等几种形式，但在实际应用中一般采用3DES方案，它还有两个别名——Triple DES和DESede。

DESede算法将密钥长度增至112位或168位，抗穷举攻击的的能力显著增强。但究其核心仍是DES算法，虽然增加迭代次数提高了安全性，但与此同时也造成处理速度较慢，密钥计算时间加长，加密效率不高的问题。

Java 6支持DES算法实现，支持112位或168位密钥长度；通过Bouncy Castle可将DES算法密钥延长至128位或192位。

DESede算法的种种劣势宣告DES算法改良路线失败，对称加密算法前途陷入困境，迫切需要产生新的对称加密算法，AES算法和IDEA算法呼之欲出。

由比利时人Daemen和Rijmen提出的Rijndael算法，以其密钥设置快、存储要求低、在硬件

更多资源请访问稀酷客(www.ckook.com)

实现和限制存储的条件下性能优异当选AES算法。AES算法支持3种长度的密钥，分为128位、192位和256位。基于较高的安全性，AES常被用于通用移动通信系统。

Java 6支持AES算法实现，支持128位、192位和256位3种长度密钥。但要实现256位长度密钥则需获得无政策限制权限文件（Unlimited Strength Jurisdiction Policy Files），通过Bouncy Castle可获得更多填充方式，增强AES算法安全性。

IDEA（International Data Encryption Algorithm，国际数据加密标准）算法是目前较为常用的电子邮件加密算法之一。电子邮件加密软件PGP（Pretty Good Privacy）使用了具有商业版权的IDEA算法，实现邮件加密/解密工作。

Java 6不支持IDEA算法实现，但通过Boucy Castle可获得相应实现，并支持128位长度密钥。

PBE（Password Based Encryption，基于口令加密）算法是一种基于口令的加密算法，其特点在于口令由用户自己掌管，采用随机数（这里我们叫做盐）杂凑多重加密等方法保证数据的安全性。

PBE算法并没有真正构建新的加密/解密算法，而是对我们已知的对称加密算法（如DES算法）做了包装。使用PBE算法对数据做加密/解密操作时，其实是使用了DES或AES等其他对称加密算法做了相应的操作。其密钥长度与相关PBE算法息息相关，如PBEWithMD5AndDES算法根据对应DES算法，其密钥长度为56位。

Java 6支持PBE算法实现，通过Boucy Castle可获得更多的PBE算法实现。

对称加密算法是现代密码学中不可或缺的算法分支之一，与之相对的非对称加密算法虽然在安全性上大大优于对称加密算法，但却无法在加密/解密速度上与之媲美。

更多资源请访问稀酷客(www.ckook.com)

第8章

高等数据加密——非对称加密算法

我们可能没有在瑞士苏黎世银行存入巨额资产的机会，但相信大多数人都在电影中见到这样一组镜头：户主带着自己的钥匙来到银行，要求取出自己寄放的物品。银行工作人员验明户主身份后，拿出另一把钥匙同户主一起打开保险柜，将用户寄放物品取出。我们可以把这个保险柜称为“双钥保险柜”。

与上一章提到的密码日记本不同，要开启双钥保险柜同时需要两把钥匙。一把钥匙留给户主，由户主保管，我们可以把它叫做“公钥”；另一把钥匙留给银行，由银行保管，我们可以把它叫做“私钥”。

如果将上述内容引申为加密/解密应用，我们就能体会到非对称加密算法与对称加密算法之间的区别。对称加密算法仅有一个密钥，既可用于加密，亦可用于解密。若完成加密/解密操作，仅需要一个密钥即可；而非对称加密算法拥有两个密钥，一个用于加密，另一个则用于解密。若要完成加密/解密操作，需要两个密钥同时参与。

相比与对称加密算法的单钥体系，非对称加密算法的双钥体系就更为安全。但非对称加密的缺点是加解密速度要远远慢于对称加密，在某些极端情况下，非对称加密算法甚至比非对称加密要慢1000倍。

8.1 非对称加密算法简述

非对称加密算法与对称加密算法的主要差别在于非对称加密算法用于加密和解密的密钥不相同，一个公开，称为公钥；一个保密，称为私钥。因此，非对称密码算法也称为双钥或公钥加密算法。

非对称加密算法解决了对称加密算法密钥分配问题，并极大地提高了算法安全性。多种B2C或B2B应用均使用非对称加密算法作为数据加密的核心算法。

8.1.1 非对称加密算法的由来

密钥管理是对称加密算法系统不容忽视的问题，它成为安全系统中最为薄弱的环节。为了弥补这一弱势，非对称加密算法应运而生。

更多资源请访问稀酷客(www.ckook.com)

1976年，W.Diffie和M.Hellman在IEEE（Institute of Electrical and Electronics Engineers，美国电气和电子工程师协会）的刊物上发表《密码学的新方向》一文，首次提出非对称加密算法。

非对称加密算法有别于对称加密算法，将密钥一分为二，公钥公开，私钥保密。公钥通过非安全通道发放，私钥则由发放者保留。公钥与私钥相对应，成对出现。公钥加密的数据，只可使用私钥对其解密。反之，私钥加密的数据，只可使用公钥对其解密。

非对称加密算法与对称加密算法相比，密钥管理问题不复存在，在安全性上有着无法逾越的高度，但却无法回避加密/解密效率低这一问题。因此，非对称加密算法往往应用在一些安全性要求相当高的领域，如B2C、B2B等电子商务平台。

注意 针对非对称加密算法的低效问题，各密码学机构主张将对称加密算法与非对称加密算法相结合，使用对称加密算法为数据加密/解密，使用公钥和私钥为对称加密算法密钥加密/解密。利用对称加密算法的高效性，加之非对称加密算法的密钥管理，提升整体加密系统的安全性。在算法设计上，非对称加密算法对待加密数据长度有着极为苛刻的要求。例如，RSA算法要求待加密数据不得大于53个字节。非对称加密算法主要用于交换对称加密算法的秘密密钥，而非数据交换。

8.1.2 非对称加密算法的家谱

非对称加密算法源于DH算法（Diffie-Hellman，密钥交换算法），由W.Diffie和M.Hellman共同提出。该算法为非对称加密算法奠定了基础，堪称非对称加密算法之鼻祖。

DH算法提出后，国际上相继出现了各种实用性更强的非对称加密算法，其构成主要是基于数学问题的求解，主要分为两类：

❑ 基于因子分解难题

RSA算法是最为典型的非对称加密算法，该算法由美国麻省理工学院（MIT）的Ron Rivest、AdiShamir和Leonard Adleman三位学者提出，并以这三位学者的姓氏开头字母命名，称为RSA算法。RSA算法是当今应用范围最为广泛的非对称加密算法，也是第一个既能用于数据加密也能用于数字签名的算法。

❑ 基于离散对数难题

ElGamal算法由Taher ElGamal提出，以自己的名字命名。该算法即可用于加密/解密，也可用于数字签名，并为数字签名算法形成标准提供参考。美国的DSS（Digital Signature Standard，数据签名标准）的DSA（Digital Signature Algorithm，数字签名算法）经ElGamal算法演变而来。

ECC（Elliptical Curve Cryptography，椭圆曲线加密）算法以椭圆曲线理论为基础，在创建密钥时可做到更快、更小，并且更有效。ECC 算法通过椭圆曲线方程式的性质产生密钥，而不是采用传统的方法利用大质数的积来产生。

目前，在Java 6中仅提供了DH和RSA两种算法实现。通过Boucy Castle可以获得ElGamal算法支持。但对于ECC算法，尤其是ECC加密算法，目前还没有相关开源组件提供相应支持。

更多资源请访问稀酷客(www.ckook.com)

8.2 密钥交换算法——DH

对称加密算法提高数据安全性的同时带来了密钥管理的复杂性。消息收发双方若想发送加密消息，必须事先约定好加密算法并发放密钥。而如何安全的传递密钥，成为对称加密算法的一个棘手问题。密钥交换算法（Diffie-Hellman算法，简称DH算法）成为解决这一问题的金钥匙！

8.2.1 简述

1976年，W.Diffie和M.Hellman在发表的论文中提出了公钥加密算法思想，但当时并没有给出具体的实施方案，原因在于没有找到单向函数（也就是消息摘要算法），但在该文中给出了通信双方通过信息交换协商密钥的算法，即Diffie-Hellman密钥交换算法（通常简称为DH算法）。该算法的目的在于让消息的收发双方可以在安全的条件下交换密钥，以备后续加密/解密使用。因此，DH算法是第一个密钥协商算法，但仅能用于密钥分配，不能用于加密或解密消息。

DH密钥交换算法的安全性基于有限域上的离散对数难题。基于这种安全性，通过DH算法进行密钥分配，使得消息的收发双方可以安全地交换一个秘密密钥，再通过这个密钥对数据进行加密和解密处理。

8.2.2 模型分析

我们仍以消息传递模型为例，甲方作为发送者，乙方作为接收者，分述甲乙双方如何构建密钥、交互密钥和加密数据。

首先，甲乙双方需要在收发消息前构建自己的密钥对，如图8-1所示。

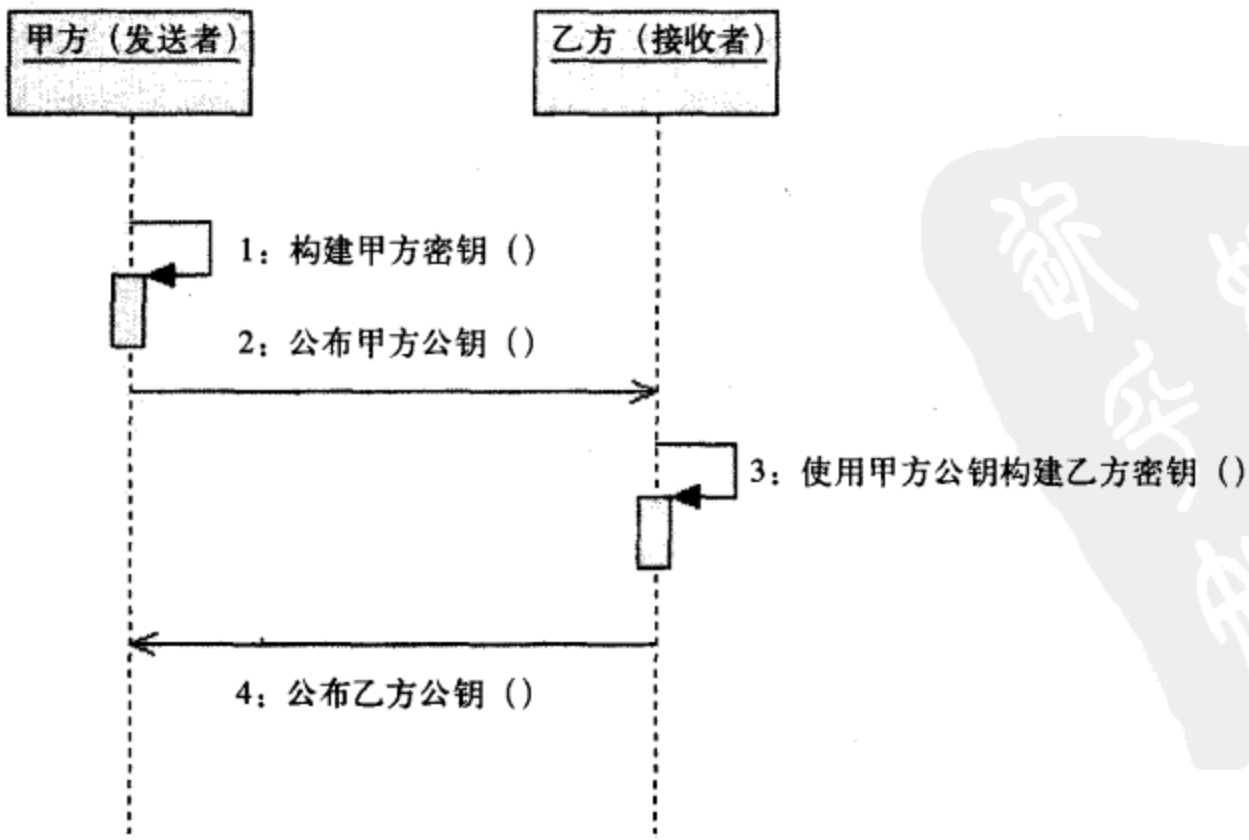


图8-1 初始化DH算法密钥对

更多资源请访问稀酷客(www.ckook.com)

甲乙两方构建密钥需要经过以下几个步骤：

1）由消息发送的一方构建密钥，这里由甲方构建密钥。

2）由构建密钥的一方向对方公布其公钥，这里由甲方向乙方发布公钥。

3）由消息接收的一方通过对方公钥构建自身密钥，这里由乙方使用甲方公钥构建乙方密钥。

4）由消息接收的一方向对方公布其公钥，这里由乙方向甲方公布公钥。

这里要注意的是，乙方构建自己的密钥对的时候需要使用甲方公钥作为参数。这是很关键的一点，如果缺少了这一环节则无法确保甲乙双方获得同一个秘密密钥，消息加密更无从谈起。

其次，假设甲乙双方事先约定好了用于数据加密的对称加密算法（如AES算法），并构建本地密钥（即对称加密算法中的秘密密钥），如图8-2和图8-3所示。

甲方需要使用自己的私钥和乙方的公钥才能构建本地密钥，即用于数据加密/解密操作的对称加密算法的秘密密钥。

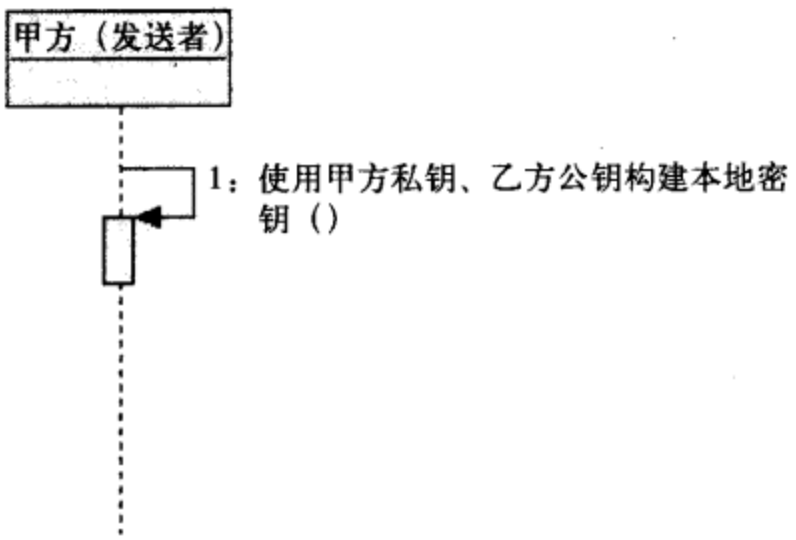


图8-2　甲方构建DH算法本地密钥

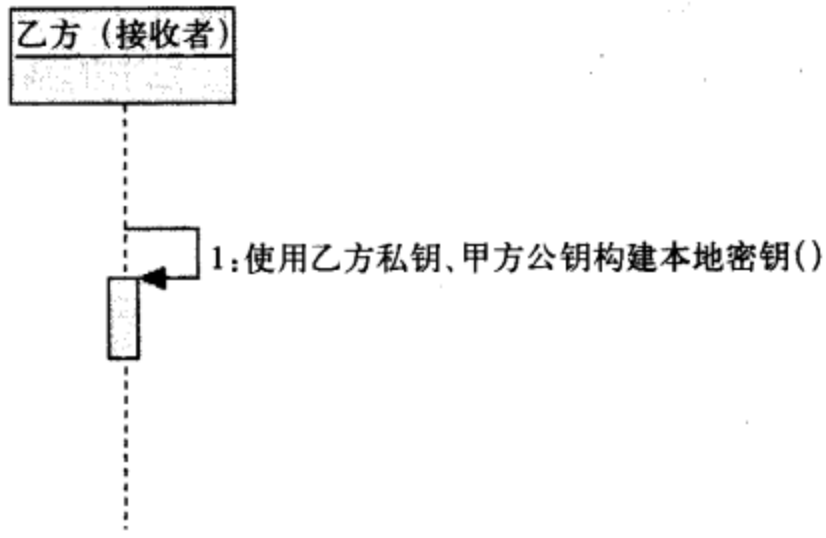


图8-3　乙方构建DH算法本地密钥

乙方构建本地密钥的方式与甲方相类似，同样需要通过使用自己的私钥和甲方的公钥构建本地密钥。

虽然，甲乙两方使用了不同的密钥来构建本地密钥，但甲乙两方得到密钥其实是一致的，我们可以通过后续的测试用例验证这一点。也正因于此，甲乙双方才能顺利地进行加密消息的传递。

最后，甲乙双方构建了本地密钥后，可按基于对称加密算法的消息传递模型完成消息传递，如图8-4所示。

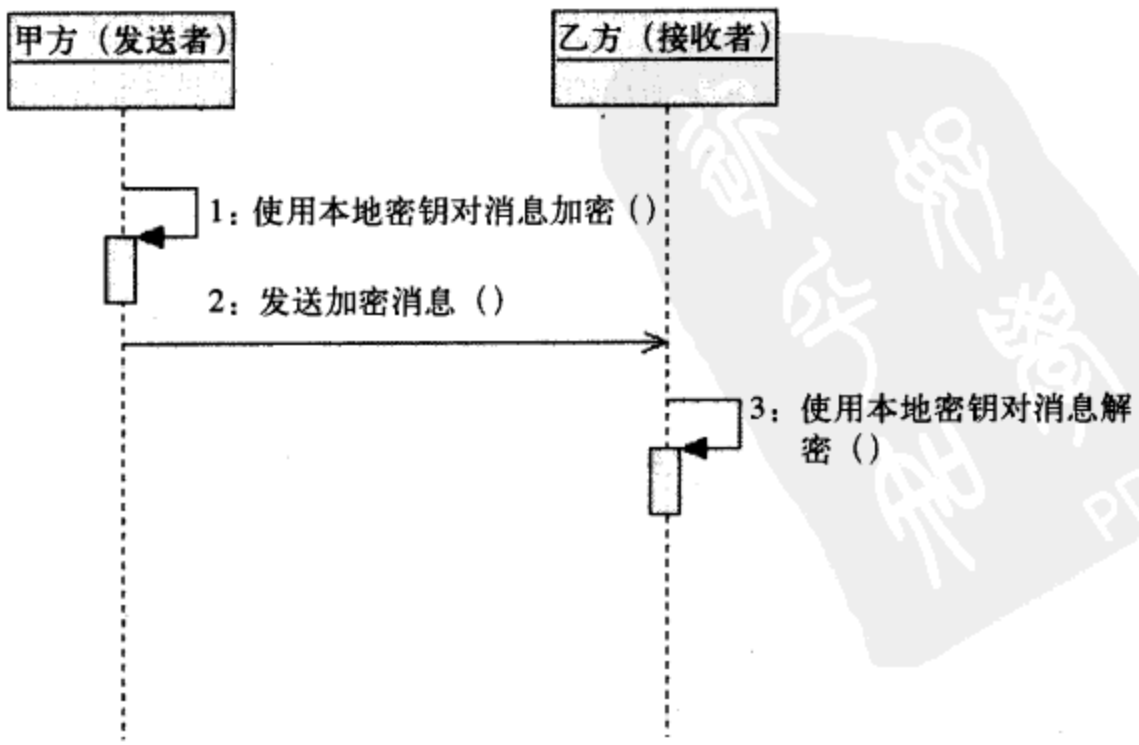


图8-4　DH算法加密消息传递

作为对称加密体制向非

更多资源请访问稀酷客(www.ckook.com)

对称加密体制的一种过渡，DH算法仅仅比一般的对称加密算法多了密钥对的构建和本地密钥的构建这两项操作，而真正的数据加密/解密操作仍由对称加密算法完成。

8.2.3 实现

Java 6提供了DH算法的相关实现，相关Java API的知识可阅读第3章内容。作为对称加密算法向非对称加密算法的一种过渡，DH算法的实现是我们了解非对称加密算法的最佳途径。

有关DH算法的Java 6实现细节如表8-1所示。

表8-1 DH算法

算法	密钥长度	密钥长度默认值	工作模式	填充方式	备注
DH	512至1024位（密钥长度为64的倍数，范围在512～1024位之间）	1024	无	无	Java 6实现

在这里，作者与读者朋友做一个约定：所有非对称算法实现的公有方法均不使用java.security和javax.crypto包及其子包中的接口或类作为参数或返回值。

实现DH算法密钥需要用到密钥对及密钥对生成器，如代码清单8-1所示。

代码清单8-1 构建DH算法甲方密钥对

```
// 实例化密钥对生成器
KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("DH");
// 初始化密钥对生成器
keyPairGenerator.initialize(1024);
// 生成密钥对
KeyPair keyPair = keyPairGenerator.generateKeyPair();
// 公钥
PublicKey publicKey = keyPair.getPublic();
// 私钥
PrivateKey privateKey = keyPair.getPrivate();
```

如果有必要，我们需要使用DH算法专用公钥/私钥接口（DHPublicKey/DHPrivateKey）强行转换上述密钥。

上述代码完成了甲方密钥的构建，要构建乙方密钥需要使用甲方公钥，具体实现如代码清单8-2所示。

代码清单8-2 构建DH算法乙方密钥对

```
// 由甲方公钥构建乙方密钥
DHParameterSpec dhParamSpec = ((DHPublicKey) pubKey).getParams();
// 实例化密钥对生成器
KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance(keyFactory.getAlgorithm());
// 初始化密钥对生成器
keyPairGenerator.initialize(dhParamSpec);
// 生成密钥对
KeyPair keyPair = keyPairGenerator.generateKeyPair();
```

更多资源请访问稀酷客(www.ckook.com)

```
// 公钥
PublicKey publicKey = keyPair.getPublic();
// 私钥
PrivateKey privateKey = keyPair.getPrivate();
```

按照作者与读者朋友的约定，密钥仅能以二进制编码形式出现。因此，我们通过Map对象封装密钥，如代码清单8-3所示。

代码清单8-3 封装DH算法密钥

```
// 将密钥对存储在Map中
Map<String, Object> keyMap = new HashMap<String, Object>(2);
keyMap.put("DHPublicKey", publicKey);
keyMap.put("DHPrivateKey", privateKey);
```

我们将在后续的initKey()方法中见到上述代码。

如果要将密钥材料转换为密钥对象，可参考代码清单8-4。

代码清单8-4 转换密钥材料

```
// 实例化密钥工厂
KeyFactory keyFactory = KeyFactory.getInstance("DH");
// 初始化公钥
// 公钥密钥材料转换
X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(publicKey);
// 产生公钥
PublicKey publicKey = keyFactory.generatePublic(x509KeySpec);
// 初始化私钥
// 私钥密钥材料转换
PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(privateKey);
// 产生私钥
PrivateKey privateKey = keyFactory.generatePrivate(pkcs8KeySpec);
```

完成甲乙方密钥的构建操作后，我们便可以完成本地密钥的构建。这里，要求使用不对称的公钥和私钥来构建本地密钥，即使用甲方私钥和乙方公钥构建甲方本地密钥；使用乙方私钥和甲方公钥构建乙方本地密钥。相关实现如代码清单8-5所示。

代码清单8-5 构建本地密钥

```
// 实例化
KeyAgreement keyAgree = KeyAgreement.getInstance(keyFactory.getAlgorithm());
// 初始化
keyAgree.init(priKey);
keyAgree.doPhase(pubKey, true);
// 生成本地密钥
SecretKey secretKey = keyAgree.generateSecret("AES");
```

完成了上述准备后，我们接下来仅仅需要使用本地密钥进行加密/解密操作了。本地密钥就是对称加密算法中的秘密密钥，对于对称加密算法的加密/解密操作，我们在第7章中已经详尽描述，就不在此复述了。完整的代码实现如代码清单8-6所示。

更多资源请访问稀酷客(www.ckook.com)

代码清单8-6 DH算法实现

```
import java.security.Key;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PublicKey;
import java.security.PrivateKey;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.util.HashMap;
import java.util.Map;
import javax.crypto.Cipher;
import javax.crypto.KeyAgreement;
import javax.crypto.SecretKey;
import javax.crypto.interfaces.DHPrivateKey;
import javax.crypto.interfaces.DHPublicKey;
import javax.crypto.spec.DHParameterSpec;
import javax.crypto.spec.SecretKeySpec;
/**
 * DH安全编码组件
 * @author 梁栋
 * @version 1.0
 */
public abstract class DHCoder {
    // 非对称加密密钥算法
    public static final String KEY_ALGORITHM = "DH";
    /**
     * 本地密钥算法，即对称加密密钥算法，
     * 可选DES、DESede和AES算法
     */
    public static final String SECRET_ALGORITHM = "AES";
    /**
     * 密钥长度
     * DH算法默认密钥长度为1024
     * 密钥长度必须是64的倍数，其范围在512位到1024位之间。
     */
    private static final int KEY_SIZE = 512;
    // 公钥
    private static final String PUBLIC_KEY = "DHPublicKey";
    // 私钥
    private static final String PRIVATE_KEY = "DHPrivateKey";
    /**
     * 初始化甲方密钥
     * @return Map 甲方密钥Map
```

更多资源请访问稀酷客(www.ckook.com)

```
     * @throws Exception
     */
    public static Map<String, Object> initKey() throws Exception {
            // 实例化密钥对生成器
            KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance(KEY_ALGORITHM);
            // 初始化密钥对生成器
            keyPairGenerator.initialize(KEY_SIZE);
            // 生成密钥对
            KeyPair keyPair = keyPairGenerator.generateKeyPair();
            // 甲方公钥
            DHPublicKey publicKey = (DHPublicKey) keyPair.getPublic();
            // 甲方私钥
            DHPrivateKey privateKey = (DHPrivateKey) keyPair.getPrivate();
            // 将密钥对存储在Map中
            Map<String, Object> keyMap = new HashMap<String, Object>(2);
            keyMap.put(PUBLIC_KEY, publicKey);
            keyMap.put(PRIVATE_KEY, privateKey);
            return keyMap;
    }
    /**
     * 初始化乙方密钥
     * @param key 甲方公钥
     * @return Map 乙方密钥Map
     * @throws Exception
     */
    public static Map<String, Object> initKey(byte[] key) throws Exception {
            // 解析甲方公钥
            // 转换公钥材料
            X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(key);
            // 实例化密钥工厂
            KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
            // 产生公钥
            PublicKey pubKey = keyFactory.generatePublic(x509KeySpec);
            // 由甲方公钥构建乙方密钥
            DHParameterSpec dhParamSpec = ((DHPublicKey) pubKey).getParams();
            // 实例化密钥对生成器
            KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance(keyFactory.getAlgorithm());
            // 初始化密钥对生成器
            keyPairGenerator.initialize(dhParamSpec);
            // 产生密钥对
            KeyPair keyPair = keyPairGenerator.genKeyPair();
            // 乙方公钥
            DHPublicKey publicKey = (DHPublicKey) keyPair.getPublic();
            // 乙方私钥
            DHPrivateKey privateKey = (DHPrivateKey) keyPair.getPrivate();
            // 将密钥对存储在Map中
            Map<String, Object> keyMap = new HashMap<String, Object>(2);
            keyMap.put(PUBLIC_KEY, publicKey);
```

更多资源请访问稀酷客(www.ckook.com)

```
        keyMap.put(PRIVATE_KEY, privateKey);
        return keyMap;
    }
    /**
     * 加密
     * @param data 待加密数据
     * @param key 密钥
     * @return byte[] 加密数据
     * @throws Exception
     */
    public static byte[] encrypt(byte[] data, byte[] key) throws Exception {
        // 生成本地密钥
        SecretKey secretKey = new SecretKeySpec(key, SECRET_ ALGORITHM);
        // 数据加密
        Cipher cipher = Cipher.getInstance(secretKey.getAlgorithm());
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);
        return cipher.doFinal(data);
    }
    /**
     * 解密<br>
     * @param data 待解密数据
     * @param key 密钥
     * @return byte[] 解密数据
     * @throws Exception
     */
    public static byte[] decrypt(byte[] data, byte[] key) throws Exception {
        // 生成本地密钥
        SecretKey secretKey = new SecretKeySpec(key, SECRET_ALGORITHM);
        // 数据解密
        Cipher cipher = Cipher.getInstance(secretKey.getAlgorithm());
        cipher.init(Cipher.DECRYPT_MODE, secretKey);
        return cipher.doFinal(data);
    }
    /**
     * 构建密钥
     * @param publicKey 公钥
     * @param privateKey 私钥
     * @return byte[] 本地密钥
     * @throws Exception
     */
    public static byte[] getSecretKey(byte[] publicKey, byte[] privateKey)
    throws Exception {
        // 实例化密钥工厂
        KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
        // 初始化公钥
        // 密钥材料转换
        X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(publicKey);
```

更多资源请访问稀酷客(www.ckook.com)

```
        // 产生公钥
        PublicKey pubKey = keyFactory.generatePublic(x509KeySpec);
        // 初始化私钥
        // 密钥材料转换
        PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(privateKey);
        // 产生私钥
        PrivateKey priKey = keyFactory.generatePrivate(pkcs8KeySpec);
        // 实例化
        KeyAgreement keyAgree = KeyAgreement.getInstance(keyFactory.getAlgorithm());
        // 初始化
        keyAgree.init(priKey);
        keyAgree.doPhase(pubKey, true);
        // 生成本地密钥
        SecretKey secretKey = keyAgree.generateSecret(SECRET_ALGORITHM);
        return secretKey.getEncoded();
    }
    /**
     * 取得私钥
     * @param keyMap 密钥Map
     * @return byte[] 私钥
     * @throws Exception
     */
    public static byte[] getPrivateKey(Map<String, Object> keyMap) throws Exception {
        Key key = (Key) keyMap.get(PRIVATE_KEY);
        return key.getEncoded();
    }
    /**
     * 取得公钥
     * @param keyMap 密钥Map
     * @return byte[] 公钥
     * @throws Exception
     */
    public static byte[] getPublicKey(Map<String, Object> keyMap)
            throws Exception {
        Key key = (Key) keyMap.get(PUBLIC_KEY);
        return key.getEncoded();
    }
}
```

对于DH算法的测试，我们需要关注两点：一点是甲乙方的本地密钥是否相同；另一点是甲方加密后的数据，乙方是否能够解析，反之亦然。

完整的DH算法测试用例如代码清单8-7所示。

代码清单8-7 DH算法实现测试用例

```
import static org.junit.Assert.*;
import java.util.Map;
import org.apache.commons.codec.binary.Base64;
```

更多资源请访问稀酷客(www.ckook.com)

```
import org.junit.Before;
import org.junit.Test;
/**
 * DH校验
 * @author 梁栋
 * @version 1.0
 */
public class DHCoderTest {
    // 甲方公钥
    private byte[] publicKey1;
    // 甲方私钥
    private byte[] privateKey1;
    // 甲方本地密钥
    private byte[] key1;
    // 乙方公钥
    private byte[] publicKey2;
    // 乙方私钥
    private byte[] privateKey2;
    // 乙方本地密钥
    private byte[] key2;
    /**
     * 初始化密钥
     * @throws Exception
     */
@Before
public final void initKey() throws Exception {
        // 生成甲方密钥对
        Map<String, Object> keyMap1 = DHCoder.initKey();
        publicKey1 = DHCoder.getPublicKey(keyMap1);
        privateKey1 = DHCoder.getPrivateKey(keyMap1);
        System.err.println("甲方公钥:\n" + Base64.encodeBase64String
        (publicKey1));
        System.err.println("甲方私钥:\n" + Base64.encodeBase64String
        (privateKey1));
        // 由甲方公钥产生本地密钥对
        Map<String, Object> keyMap2 = DHCoder.initKey(publicKey1);
        publicKey2 = DHCoder.getPublicKey(keyMap2);
        privateKey2 = DHCoder.getPrivateKey(keyMap2);
        System.err.println("乙方公钥:\n" + Base64.encodeBase64String
        (publicKey2));
        System.err.println("乙方私钥:\n" + Base64.encodeBase64String
        (privateKey2));
        key1 = DHCoder.getSecretKey(publicKey2, privateKey1);
        System.err.println("甲方本地密钥:\n" + Base64.encodeBase64String(key1));
        key2 = DHCoder.getSecretKey(publicKey1, privateKey2);
        System.err.println("乙方本地密钥:\n" + Base64.encodeBase64String(key2));
        // 校验
        assertArrayEquals(key1, key2);
```

更多资源请访问稀酷客(www.ckook.com)

```
}
/**
 * 校验
 * @throws Exception
 */
@Test
public final void test() throws Exception {
        System.err.println("\n=====甲方向乙方发送加密数据=====");
        String input1 = "密码交换算法";
        System.err.println("原文: " + input1);
        System.err.println("---使用甲方本地密钥对数据加密---");
        // 使用甲方本地密钥对数据加密
        byte[] code1 = DHCoder.encrypt(input1.getBytes(), key1);
        System.err.println("加密: " + Base64.encodeBase64String(code1));
        System.err.println("---使用乙方本地密钥对数据解密---");
        // 使用乙方本地密钥对数据解密
        byte[] decode1 = DHCoder.decrypt(code1, key2);
        String output1 = (new String(decode1));
        System.err.println("解密: " + output1);
        assertEquals(input1, output1);
        System.err.println("\n=====乙方向甲方发送加密数据=====");
        String input2 = "DH";
        System.err.println("原文: " + input2);
        System.err.println("---使用乙方本地密钥对数据加密---");
        // 使用乙方本地密钥对数据加密
        byte[] code2 = DHCoder.encrypt(input2.getBytes(), key2);
        System.err.println("加密: " + Base64.encodeBase64String(code2));
        System.err.println("---使用甲方本地密钥对数据解密---");
        // 使用甲方本地密钥对数据解密
        byte[] decode2 = DHCoder.decrypt(code2, key1);
        String output2 = (new String(decode2));
        System.err.println("解密: " + output2);
        // 校验
        assertEquals(input2, output2);
    }
}
```

按照惯例，我们使用Base64算法对密钥编码，在控制台得到相关信息，如以下代码所示：

```
甲方公钥:
MIHgMIGXBgkqhkiG9w0BAwEwgYkCQQD8poLOjhLKuibvzPcRDlJtsHiwXt7LzR60ogjzrhYXrgHz
W5Gkfm32NBPF4S7QiZvNEyrNUNmRUb3EPuc3WS4XAkBnhHGyepz0TukaScUUfbGpqvJE8FpDTWSG
kx0tFCcbnjUDC3H9c9oXkGmzLik1Yw4cIGI1TQ2iCmxBblC+eUykAgIBgANEAAJBAJei+kbWCpbA
EiXXcMrj991qHahR6nSQ4EfKqBah4+apd+zV92iy1C+rwsv8ea2V43zhesySZcylPnosfBl7haA=
甲方私钥:
MIHRAgEAMIGXBgkqhkiG9w0BAwEwgYkCQQD8poLOjhLKuibvzPcRDlJtsHiwXt7LzR60ogjzrhYX
rgHzW5Gkfm32NBPF4S7QiZvNEyrNUNmRUb3EPuc3WS4XAkBnhHGyepz0TukaScUUfbGpqvJE8FpD
TWSGkx0tFCcbnjUDC3H9c9oXkGmzLik1Yw4cIGI1TQ2iCmxBblC+eUykAgIBgAQyAjA1b4WOjC0x
efoTnRX9Y7cZ5EhVh2lqaOZxdaHMA1U4LMKQZQmNIewW796UkVWeW9o=
```

更多资源请访问稀酷客(www.ckook.com)

```
乙方公钥：
MIHfMIGXBgkqhkiG9w0BAwEwgYkCQQD8poLOjhLKuibvzPcRDlJtsHiwXt7LzR60ogjzrhYXrgHz
W5Gkfm32NBPF4S7QiZvNEyrNUNmRUb3EPuc3WS4XAkBnhHGyepz0TukaScUUfbGpqvJE8FpDTWSG
kx0tFCcbnjUDC3H9c9oXkGmzLik1Yw4cIGI1TQ2iCmxBblC+eUykAgIBgANDAAJAEdxKB72W1Sgq
/QRhh7WOXS31arbo55hEdLDn9vL5PT26rlaEQ5i2wECCJ0zXHIXInQHaxbftwP734Ar/vvMMBQ==
乙方私钥：
MIHSAgEAMIGXBgkqhkiG9w0BAwEwgYkCQQD8poLOjhLKuibvzPcRDlJtsHiwXt7LzR60ogjzrhYX
rgHzW5Gkfm32NBPF4S7QiZvNEyrNUNmRUb3EPuc3WS4XAkBnhHGyepz0TukaScUUfbGpqvJE8FpD
TWSGkx0tFCcbnjUDC3H9c9oXkGmzLik1Yw4cIGI1TQ2iCmxBblC+eUykAgIBgAQzAjEAog1vG/Nm
PHuVfVUHe53MkO1jTNDXYFPpY92+G6BCEzeweNDEoRj8EVQdClf7xz+t
```

不论公钥还是私钥，不论甲方还是乙方对密钥的长度都难以接受。若作者选择1024位的密钥长度，控制台输出的密钥信息打印出来恐怕要够两页A4纸了。

甲乙方的本地密钥是否相同呢？我们将其通过Base64编码，在控制台中得到如下信息：

```
甲方本地密钥：
FNDVElg+KyfttpT81YWHsiaTYNs2e0fzhf8OwCDWUGs=
乙方本地密钥：
FNDVElg+KyfttpT81YWHsiaTYNs2e0fzhf8OwCDWUGs=
```

显然，甲乙方的本地密钥是一致的。

接下来验证甲方加密的数据乙方是否可以解密，在控制台中得到如下信息：

```
=====甲方向乙方发送加密数据=====
原文：密码交换算法
---使用甲方本地密钥对数据加密---
加密：IyzI6a6HyEq4927UoHQ8jIgMrA9RFtKf2Hcd/503eTM=
---使用乙方本地密钥对数据解密---
解密：密码交换算法
```

反之，验证乙方加密的数据甲方是否可以解密，在控制台中得到如下信息：

```
=====乙方向甲方发送加密数据=====
原文：DH
---使用乙方本地密钥对数据加密---
加密：OVvejG+sj98I7BIqLGWpmA==
---使用甲方本地密钥对数据解密---
解密：DH
```

通过上述测试，我们可以验证：经由甲乙双方构建的秘密密钥相同，基于DH算法实现的加密通信系统实际上是使用同一个秘密密钥完成相应的加密/解密对称加密系统。

对于DH算法而言，算法的安全强度在于密钥长度和对称加密算法的选择。

- DH算法支持的密钥长度为64的倍数位，取值范围在512～1024位（含1024位）之间。密钥长度与加密安全强度成正比，与运算时间成反比。在使用时，需选择合适的密钥长度。
- 对称加密算法可以选择DES、DESede和AES算法等。

合理选择密钥长度和对称加密算法是构建基于DH算法密码系统的关键。

更多资源请访问稀酷客(www.ckook.com)

8.3 典型非对称加密算法——RSA

DH算法的诞生为后续非对称加密算法奠定了基础，较为典型的对称加密算法（如ElGamal、RSA和ECC算法等）都是在DH算法提出后相继出现的，并且其算法核心都源于数学问题。

RSA算法基于大数因子分解难题，而ElGamal算法和ECC算法则是基于离散对数难题。

目前，各种主流计算机语言都提供了对RSA算法的支持。在Java 6中，我们可以很方便地构建该算法。

8.3.1 简述

1978年，美国麻省理工学院（MIT）的Ron Rivest、Adi Shamir和Leonard Adleman三位学者提出了一种新的非对称加密算法，这种算法以这三位学者的姓氏开头字母命名，被称为RSA算法。

RSA算法是唯一被广泛接受并实现的通用公开加密算法，目前已经成为非对称加密算法国际标准。不仅如此，RSA算法既可用于数据加密也可用于数字签名。我们熟知的电子邮件加密软件PGP就采用了RSA算法对数据进行加密/解密和数字签名处理。

RSA算法将非对称加密算法推向了一个高潮，但它并不是没有缺点。相比DES以及其他对称加密算法而言，RSA算法要慢得多。作为加密算法，尤其是作为非对称加密算法的典型，针对RSA算法的破解自诞生之日起就从未停止过。

1999年，RSA-155（512位）被成功分解，历时5个月。

2002年，RSA-158也被成功因数分解。

当然，基于大数因子分解数学难题的非对称加密算法仅有RSA算法这一种，但这不代表非对称加密算法除了RSA算法后继无人。基于离散对数问题的非对称加密算法包括ElGamal和ECC这两种算法，它们同样是优秀的非对称加密算法。

8.3.2 模型分析

我们继续以消息传递模型为例，阐述基于RSA算法的消息传递模型是如何工作的。

通过了解基于DH算法的消息传递模型，我们更容易理解基于非对称加密算法的消息传递模型的工作方式。或者说，基于DH算法的消息传递模型更为底层。

RSA算法代表了真正的非对称加密算法，其操作较之DH算法较为简单。在构建密钥对方面，RSA算法就相当简单，如图8-5所示。

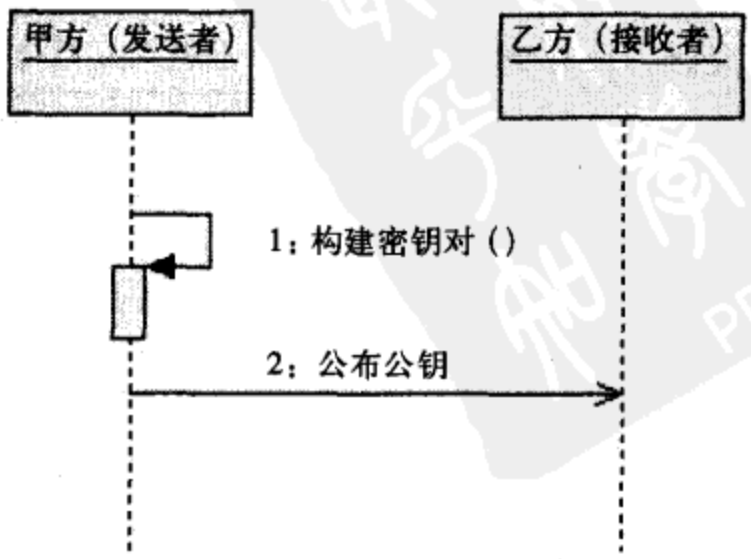


图8-5 构建RSA算法密钥对

在图8-5中，我们仍以甲乙两方收发消息为例。甲方作为消息的发送方，乙方作为消息的接收方。我们假设甲乙双方在消息传递之前已将RSA算法

更多资源请访问稀酷客(www.ckook.com)

作为消息传递的加密算法。为完成加密消息传递，甲乙双方需要以下操作：

1）由消息发送的一方构建密钥对，这里由甲方完成。

2）由消息发送的一方公布公钥至消息接收方，这里由甲方将公钥公布给乙方。

完成这两步操作后，甲乙双方就可以进行加密消息传递了，如图8-6和图8-7所示。

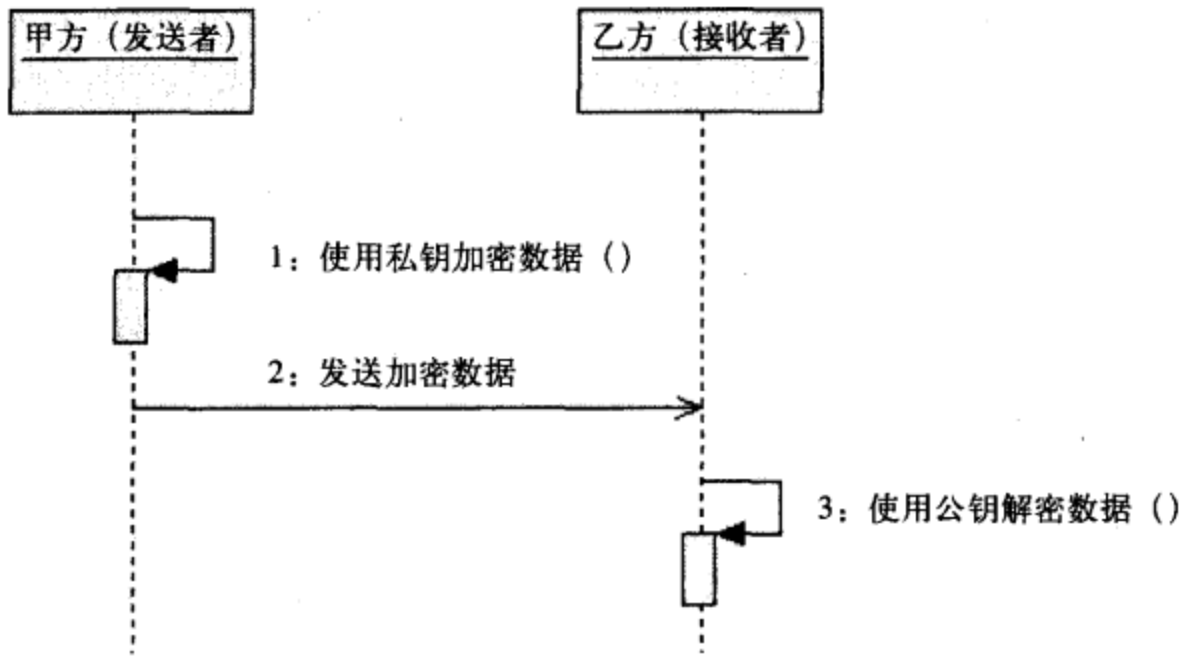


图8-6 甲方向乙方发送RSA算法加密数据

这里需要读者朋友注意：甲方向乙方发送数据时，使用私钥对数据做加密处理；乙方接收到加密数据后，使用公钥对数据做解密处理。

我们需要注意，在非对称加密算法领域中，对于私钥加密的数据，只能使用公钥解密。简言之，“私钥加密，公钥解密”。

相对于“私钥加密，公钥解密”的实现，RSA算法提供了另一种加密/解密方式：“公钥加密，私钥解密”。这使得乙方可以使用公钥加密数据，如图8-7所示。

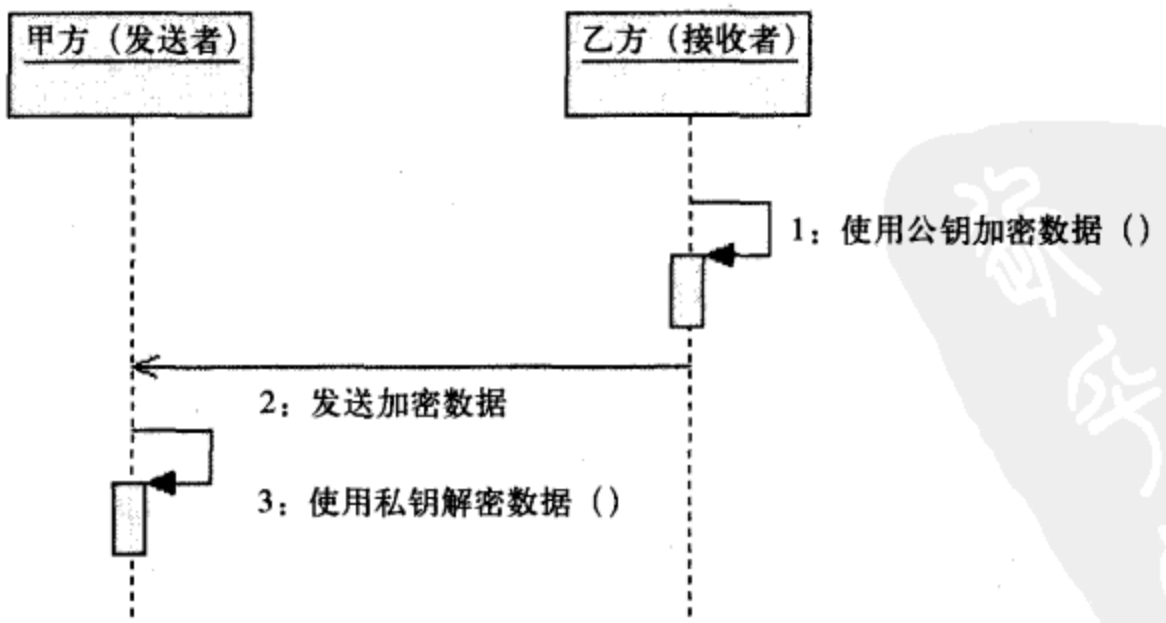


图8-7 乙方向甲方发送RSA算法加密数据

甲方可以向乙方发送加密消息，乙方也同样可以向甲方发送加密消息。在图8-7中，乙方使用公钥加密消息，甲方则使用私钥对加密消息解密。这是一种“公钥加密，私钥解密”的加密/解密方式。

更多资源请访问稀酷客(www.ckook.com)

用公钥加密数据的方式是否可取呢？公钥是通过甲方发送给乙方的，其在传递过程中很有可能被截获，也就是说窃听者很有可能获得公钥。如果窃听者获得了公钥，向甲方发送数据，甲方是无法辨别消息的真伪的。因此，虽然可以使用公钥对数据加密，但这种方式还是会有存在一定的安全隐患。如果要建立更安全的加密消息传递模型，就需要甲乙两方构建两套非对称加密算法密钥，仅遵循“私钥加密，公钥解密”的方式进行加密消息传递。

8.3.3　实现

Java 6提供了RSA算法实现，相关内容请读者朋友阅读第3章。RSA算法在理解上较之DH算法更为简单，在实现层面上也同样如此。

RSA算法与DH算法在密钥管理和加密/解密两方面有所不同：一方面，甲方保留了私钥，而将公钥公布于乙方，甲乙双方密钥一一对应；另一方面，私钥用于解密，公钥则用于加密，反之亦然。

这里要注意一点，在RSA算法中，公钥既可用于解密也可用于加密，私钥也是如此。但公钥加密的数据只能用私钥对其解密，而私钥加密的数据只能用公钥对其解密，这种对应关系是不能违背的。

RSA算法实现较之DH算法实现较为简单，大部分内容与DH算法较为接近。与DH算法不同的是，RSA算法仅需要一套密钥即可完成加密/解密操作。Bouncy Castle对RSA算法做了相应扩充，增加了ISO9796-1Padding填充方式。

有关RSA算法的Java 6实现与Bouncy Castle实现细节如表8-2所示。

表8-2　RSA算法

算法	密钥长度	密钥长度默认值	工作模式	填充方式	备注
RSA	512～65536位（密钥长度必须是64的倍数）	1024	ECB	NoPadding、PKCS1Padding、OAEPWITHMD5AndMGF1Padding、OAEPWITHSHA1AndMGF1Padding、OAEPWITHSHA256AndMGF1Padding、OAEPWITHSHA384AndMGF1Padding、OAEPWITHSHA512AndMGF1Padding	Java 6实现
		2048	NONE	NoPadding、PKCS1Padding、OAEPWithMD5AndMGF1Padding、OAEPWithSHA1AndMGF1Padding、OAEPWithSHA224AndMGF1Padding、OAEPWithSHA256AndMGF1Padding、OAEPWithSHA384AndMGF1Padding、OAEPWithSHA512AndMGF1Padding、ISO9796-1Padding	Bouncy Castle实现

更多资源请访问稀酷客(www.ckook.com)

完整实现如代码清单8-8所示。

代码清单8-8 RSA算法实现

```
import java.security.Key;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.util.HashMap;
import java.util.Map;
import javax.crypto.Cipher;
/**
 * RSA安全编码组件
 * @author 梁栋
 * @version 1.0
 */
public abstract class RSACoder {
    // 非对称加密密钥算法
    public static final String KEY_ALGORITHM = "RSA";
    // 公钥
    private static final String PUBLIC_KEY = "RSAPublicKey";
    // 私钥
    private static final String PRIVATE_KEY = "RSAPrivateKey";
    /**
     * RSA密钥长度
     * 默认1024位,
     * 密钥长度必须是64的倍数,
     * 范围在512～65536位之间。
     */
    private static final int KEY_SIZE = 512;
    /**
     * 私钥解密
     * @param data 待解密数据
     * @param key 私钥
     * @return byte[] 解密数据
     * @throws Exception
     */
    public static byte[] decryptByPrivateKey(byte[] data, byte[] key) throws Exception {
            // 取得私钥
            PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(key);
            KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
            // 生成私钥
```

更多资源请访问稀酷客(www.ckook.com)

```
        PrivateKey privateKey = keyFactory.generatePrivate(pkcs8KeySpec);
        // 对数据解密
        Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
        cipher.init(Cipher.DECRYPT_MODE, privateKey);
        return cipher.doFinal(data);
    }
    /**
     * 公钥解密
     * @param data 待解密数据
     * @param key 公钥
     * @return byte[] 解密数据
     * @throws Exception
     */
    public static byte[] decryptByPublicKey(byte[] data, byte[] key) throws Exception {
        // 取得公钥
        X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(key);
        KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
        // 生成公钥
        PublicKey publicKey = keyFactory.generatePublic(x509KeySpec);
        // 对数据解密
        Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
        cipher.init(Cipher.DECRYPT_MODE, publicKey);
        return cipher.doFinal(data);
    }
    /**
     * 公钥加密
     * @param data 待加密数据
     * @param key 公钥
     * @return byte[] 加密数据
     * @throws Exception
     */
    public static byte[] encryptByPublicKey(byte[] data, byte[] key) throws Exception {
        // 取得公钥
        X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(key);
        KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
        PublicKey publicKey = keyFactory.generatePublic(x509KeySpec);
        // 对数据加密
        Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
        cipher.init(Cipher.ENCRYPT_MODE, publicKey);
        return cipher.doFinal(data);
    }
    /**
     * 私钥加密
     * @param data 待加密数据
     * @param key 私钥
     * @return byte[] 加密数据
     * @throws Exception
     */
```

更多资源请访问稀酷客(www.ckook.com)

```
public static byte[] encryptByPrivateKey(byte[] data, byte[] key) throws Exception {
        // 取得私钥
        PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(key);
        KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
        // 生成私钥
        PrivateKey privateKey = keyFactory.generatePrivate(pkcs8KeySpec);
        // 对数据加密
        Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
        cipher.init(Cipher.ENCRYPT_MODE, privateKey);
        return cipher.doFinal(data);
}
/**
 * 取得私钥
 * @param keyMap 密钥Map
 * @return byte[] 私钥
 * @throws Exception
 */
public static byte[] getPrivateKey(Map<String, Object> keyMap) throws Exception {
        Key key = (Key) keyMap.get(PRIVATE_KEY);
        return key.getEncoded();
}
/**
 * 取得公钥
 * @param keyMap 密钥Map
 * @return byte[] 公钥
 * @throws Exception
 */
public static byte[] getPublicKey(Map<String, Object> keyMap) throws Exception {
        Key key = (Key) keyMap.get(PUBLIC_KEY);
        return key.getEncoded();
}
/**
 * 初始化密钥
 * @return Map 密钥Map
 * @throws Exception
 */
public static Map<String, Object> initKey() throws Exception {
        // 实例化密钥对生成器
        KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance (KEY_ALGORITHM);
        // 初始化密钥对生成器
        keyPairGen.initialize(KEY_SIZE);
        // 生成密钥对
        KeyPair keyPair = keyPairGen.generateKeyPair();
        // 公钥
        RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();
        // 私钥
        RSAPrivateKey privateKey = (RSAPrivateKey) keyPair.getPrivate();
        // 封装密钥
```

更多资源请访问稀酷客(www.ckook.com)

```
            Map<String, Object> keyMap = new HashMap<String, Object>(2);
            keyMap.put(PUBLIC_KEY, publicKey);
            keyMap.put(PRIVATE_KEY, privateKey);
            return keyMap;
    }
}
```

RSA算法实现易于理解，对于RSA算法的测试只需要注意经公钥加密的数据是否可以通过私钥将其解密，反之，经私钥加密的数据是否可以通过公钥将其解密。完整的测试用例如代码清单8-9所示。

代码清单8-9 RSA算法实现测试用例

```
import static org.junit.Assert.*;
import org.apache.commons.codec.binary.Base64;
import org.junit.Before;
import org.junit.Test;
import java.util.Map;
/**
 * RSA校验
 * @author 梁栋
 * @version 1.0
 */
public class RSACoderTest {
    // 公钥
    private byte[] publicKey;
    // 私钥
    private byte[] privateKey;
    /**
     * 初始化密钥
     * @throws Exception
     */
    @Before
    public void initKey() throws Exception {
        // 初始化密钥
        Map<String, Object> keyMap = RSACoder.initKey();
        publicKey = RSACoder.getPublicKey(keyMap);
        privateKey = RSACoder.getPrivateKey(keyMap);
        System.err.println("公钥:\n" + Base64.encodeBase64String(publicKey));
        System.err.println("私钥: \n" + Base64.encodeBase64String(privateKey));
    }
    /**
     * 校验
     * @throws Exception
     */
    @Test
    public void test() throws Exception {
        System.err.println("\n---私钥加密--公钥解密---");
```

更多资源请访问稀酷客(www.ckook.com)

```
        String inputStr1 = "RSA加密算法";
        byte[] data1 = inputStr1.getBytes();
        System.err.println("原文:\n" + inputStr1);
        // 加密
        byte[] encodedData1 = RSACoder.encryptByPrivateKey(data1, privateKey);
        System.err.println("加密后:\n" + Base64.encodeBase64String (encodedData1));
        // 解密
        byte[] decodedData1 = RSACoder.decryptByPublicKey(encodedData1,
                publicKey);
        String outputStr1 = new String(decodedData1);
        System.err.println("解密后:\n" + outputStr1);
        // 校验
        assertEquals(inputStr1, outputStr1);
        System.err.println("\n---公钥加密--私钥解密---");
        String inputStr2 = "RSA Encypt Algorithm";
        byte[] data2 = inputStr2.getBytes();
        System.err.println("原文:\n" + inputStr2);
        // 加密
        byte[] encodedData2 = RSACoder.encryptByPublicKey(data2, publicKey);
        System.err.println("加密后:\n" + Base64.encodeBase64String (encodedData2));
        // 解密
        byte[] decodedData2 = RSACoder.decryptByPrivateKey(encodedData2,
                privateKey);
        String outputStr2 = new String(decodedData2);
        System.err.println("解密后: " + outputStr2);
        // 校验
        assertEquals(inputStr2, outputStr2);
    }
}
```

在控制台中输出了密钥的Base64编码信息，如下文所示：

```
公钥:
MFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAJ0PY1HiU439wWfWvQKMEgjjE/swks9KCG0UJy2qmSBs
2e76f0eTswnNl7nFxbBr5TRRyl4EhkudWDIr0u6kBgcCAwEAAQ==
私钥:
MIIBVQIBADANBgkqhkiG9w0BAQEFAASCAT8wggE7AgEAAkEAnQ9iUeJTjf3BZ9a9AowSCOMT+zCS
z0oIbRQnLaqZIGzZ7vp/R5OzCc2XucXFsGvlNFHKXgSGS51YMivS7qQGBwIDAQABAkAjMB4sEFP9
/PtG43KHTpB/0zhXz8MklAadQaWhcpZKEB3LRINLH4jJ2UJxGWXMDS5nZtbG3/1jYd2Ee0bwGb7B
AiEA8M/t/6MB4Yx4C4OBkd51TDByKJmKIEnFu2HSyzA6c2ECIQCm9zi9OuX2N/5rVytXfA+Oj7L1
wetSOrGbEHXX3D/aZwIhAMRHJlyr13eoj5wKlww2/vJXtmSjKPNCThl6FV8p8yphAiBVJyC44aEG
wefvtrVUGOGWQ5Nx40Sw215ZRzvSq3GlYQIhALIHY3fxmLlEg3NC269ouFsYeTF/EO+M02+pazz+
3UPv
```

与DH算法不同，RSA算法仅需要一套密钥即可完成加密/解密操作，并且我们发现公钥的密钥长度明显小于私钥的密钥长度，更便于发送和携带。

私钥加密公钥解密，如以下代码所示：

更多资源请访问稀酷客(www.ckook.com)

```
---私钥加密--公钥解密---
原文:
RSA加密算法
加密后:
EePGm+yWtFvgSvc1pmh1hNoy3KyH0gssjc2F1vPSNkFAOOFOvvVIPQAmeRtTD+L3oUKUC61zQeqf
N2B/t0ylxg==
解密后:
RSA加密算法
```

反之，公钥加密私钥解密，如以下代码所示：

```
---公钥加密--私钥解密---
原文:
RSA Encypt Algorithm
加密后:
hehNcGA8EGilNk3FJ7snGOU9XKGHN7t6DJlHQG9Ddi+h/xdk/IzWs3+SJfFsEnrTQe+96UvpEmF2
atA7+Fndgw==
解密后: RSA Encypt Algorithm
```

RSA算法公钥长度远小于私钥长度，并遵循“公钥加密，私钥解密”和“私钥加密，公钥解密”这两项加密/解密原则。

8.4 常用非对称加密算法——ElGamal

在非对称加密算法中，几乎所有的算法都是基于数学问题而建立的：RSA算法基于大数因子分解数学难题，而ElGamal算法和ECC算法则基于离散对数问题。与典型非对称加密算法RSA算法相比，ElGamal算法则被称为常用非对称加密算法。

8.4.1 简述

1985年，Taher ElGamal提出了一种基于离散对数问题的非对称加密算法，该算法即可用于加密，又可用于数字签名，是除了RSA算法外最具有代表性的公钥加密算法之一。Taher ElGamal用自己的名字定义了这种算法——ElGamal算法。

由于ElGamal算法具有较好的安全性，因此得到了广泛的应用。著名的美国数字签名标准（Digital Signature Standard，DSS）就是采用了ElGamal签名方案的一种变形——DSA（Digital Signature Algorithm）。ElGamal的一个不足之处是它的密文会成倍扩张。

8.4.2 模型分析

Java 6并没有提供ElGamal算法实现，而相关ElGamal算法的实现资料也相当有限。作者以Bouncy Castle对于ElGamal算法实现为例，描述基于ElGamal算法的消息传递模型，如图8-8所示。

ElGamal算法在构建密钥时的操作流程几乎与RSA算法完全一致。不同的是，这次密钥对的构建者换了主人。我们仍以甲乙两方收发消息为例，甲方作为消息的发送方，乙方作为消息

更多资源请访问稀酷客(www.ckook.com)

的接收方。我们假设甲乙双方在消息传递之前已将ElGamal算法作为消息传递的加密算法。为完成加密消息传递，甲乙双方需要以下操作：

1）由消息发送的一方构建密钥对，这里由乙方完成。

2）由消息发送的一方公布公钥至消息接收方，这里由乙方将公钥公布给甲方。

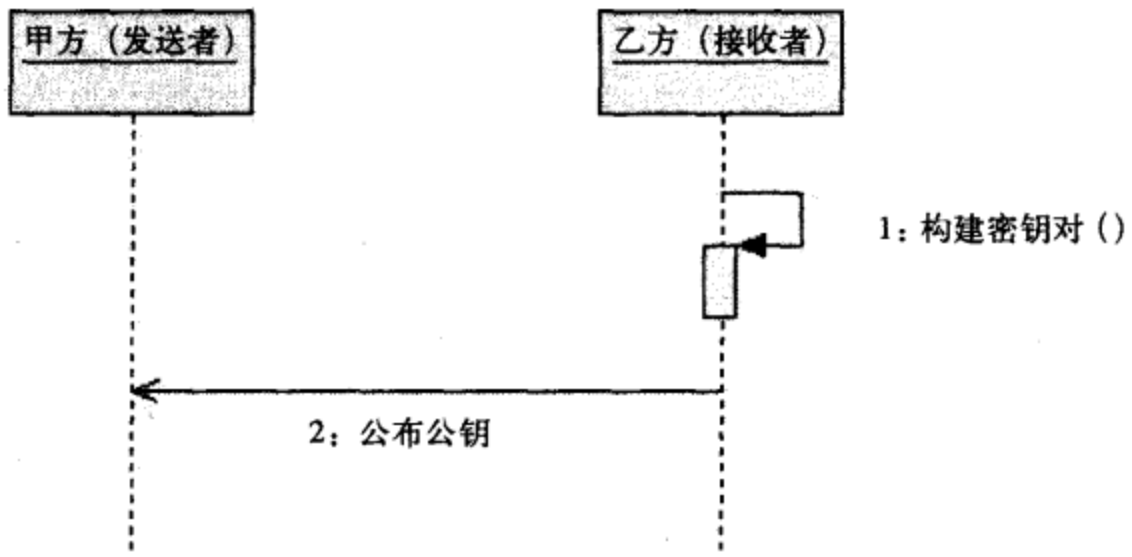


图8-8 构建ElGamal密钥对

完成这两步操作后，甲方就可以向乙方发送加密消息，如图8-9所示。

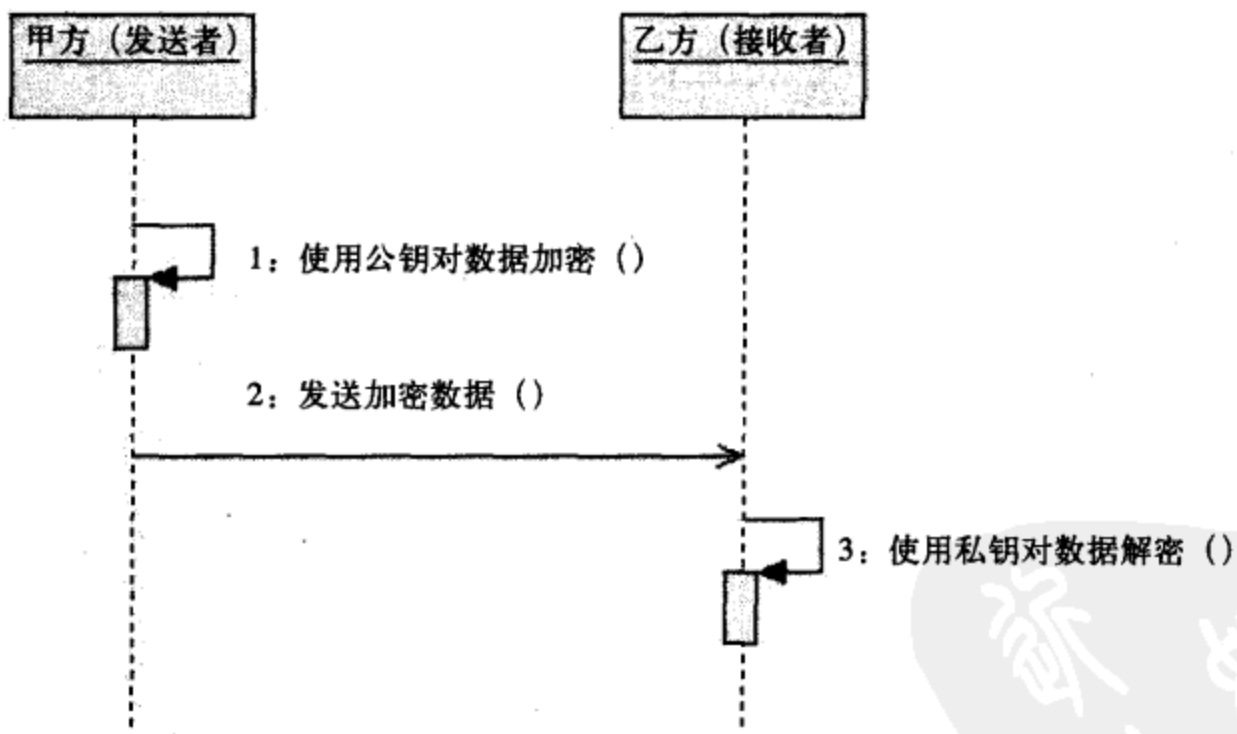


图8-9 甲方向乙方发送ElGamal加密数据

这里要注意，Bouncy Castle提供的ElGamal算法实现遵循“公钥加密，私钥解密”的加密/解密方式，由于公钥可通过非安全渠道发布给对方，其公钥数据加密的安全强度意义不大。

8.4.3 实现

很遗憾，作为常用非对称加密算法的ElGamal算法并没有出现在Java 6的API中，但却包含在了Bouncy Castle的API中，弥补了Java语言缺少对于ElGamal算法支持的缺憾。

有关ElGamal算法的Bouncy Castle实现细节如表8-3所示。

更多资源请访问稀酷客(www.ckook.com)

表8-3 ElGamal算法

算法	密钥长度	密钥长度默认值	工作模式	填充方式	备注
ElGamal	160～16 384位（密钥长度必须是8的倍数）	1024	ECB、NONE	NoPadding、PKCS1Padding、OAEPWithMD5AndMGF1Padding、OAEPWithSHA1AndMGF1Padding、OAEPWithSHA224AndMGF1Padding、OAEPWithSHA256AndMGF1Padding、OAEPWithSHA384AndMGF1Padding、OAEPWithSHA512AndMGF1Padding、ISO9796-1Padding	Bouncy Castle实现

JCE框架为其他非对称加密算法实现提供了一个构建密钥对方式，均基于DH算法参数材料——DHParameterSpec类。代码清单8-10展示了如何构建ElGamal算法密钥对。

代码清单8-10 构建ElGamal算法密钥对

```
import java.security.Security;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
// 省略
// 加入BouncyCastleProvider支持
Security.addProvider(new BouncyCastleProvider());
// 实例化算法参数生成器
AlgorithmParameterGenerator apg = AlgorithmParameterGenerator.getInstance(KEY_ALGORITHM);
// 初始化算法参数生成器
apg.init(KEY_SIZE);
// 生成算法参数
AlgorithmParameters params = apg.generateParameters();
// 构建参数材料
DHParameterSpec elParams = (DHParameterSpec) params.getParameterSpec(DHParameterSpec.class);
// 实例化密钥对生成器
KeyPairGenerator kpg = KeyPairGenerator.getInstance(KEY_ALGORITHM);
// 初始化密钥对生成器
kpg.initialize(elParams, new SecureRandom());
// 生成密钥对
KeyPair keys = kpg.genKeyPair();
```

有了密钥对实例化对象keys自然就可以获得相应的密钥了。

需要注意的是，这里我们使用Bouncy Castle提供的ElGamal算法实现，在使用各种引擎类做实例化操作前，需导入Bouncy Castle提供者，如代码清单8-11所示。

代码清单8-11 导入Bouncy Castle提供者

```
import java.security.Security;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
// 省略
// 加入BouncyCastleProvider支持
Security.addProvider(new BouncyCastleProvider());
```

更多资源请访问稀酷客(www.ckook.com)

抛开密钥对的生成实现，ElGamal算法实现与RSA算法实现几乎一致，如代码清单8-12所示。

代码清单8-12 ElGamal算法实现

```
import java.security.AlgorithmParameterGenerator;
import java.security.AlgorithmParameters;
import java.security.Key;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.SecureRandom;
import java.security.Security;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.util.HashMap;
import java.util.Map;
import javax.crypto.Cipher;
import javax.crypto.spec.DHParameterSpec;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
/**
 * ElGamal安全编码组件
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class ElGamalCoder {
    // 非对称加密密钥算法
    public static final String KEY_ALGORITHM = "ElGamal";
    /**
     * 密钥长度
     * ElGamal算法
     * 默认密钥长度为1024
     * 密钥长度范围在160～16384位不等，
     * 且密钥长度必须是8的倍数。
     */
    private static final int KEY_SIZE = 256;
    // 公钥
    private static final String PUBLIC_KEY = "ElGamalPublicKey";
    // 私钥
    private static final String PRIVATE_KEY = "ElGamalPrivateKey";
    /**
     * 用私钥解密
     * @param data 待解密数据
     * @param key 私钥
```

更多资源请访问稀酷客(www.ckook.com)

```
     * @return byte[] 解密数据
     * @throws Exception
     */
    public static byte[] decryptByPrivateKey(byte[] data, byte[] key) throws Exception {
            // 加入BouncyCastleProvider支持
            Security.addProvider(new BouncyCastleProvider());
            // 私钥材料转换
            PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(key);
            // 实例化密钥工厂
            KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
            // 生成私钥
            Key privateKey = keyFactory.generatePrivate(pkcs8KeySpec);
            // 对数据解密
            Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
            cipher.init(Cipher.DECRYPT_MODE, privateKey);
            return cipher.doFinal(data);
    }
    /**
     * 用公钥加密
     * @param data 待加密数据
     * @param key 公钥
     * @return byte[] 加密数据
     * @throws Exception
     */
    public static byte[] encryptByPublicKey(byte[] data, byte[] key) throws Exception {
            // 加入BouncyCastleProvider支持
            Security.addProvider(new BouncyCastleProvider());
            // 公钥材料转换
            X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(key);
            // 实例化密钥工厂
            KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
            // 生成公钥
            Key publicKey = keyFactory.generatePublic(x509KeySpec);
            // 对数据加密
            Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
            cipher.init(Cipher.ENCRYPT_MODE, publicKey);
            return cipher.doFinal(data);
    }
    /**
     * 生成密钥
     * @return Map 密钥Map
     * @throws Exception
     */
    public static Map<String, Object> initKey() throws Exception {
            // 加入BouncyCastleProvider支持
            Security.addProvider(new BouncyCastleProvider());
            // 实例化算法参数生成器
```

更多资源请访问稀酷客(www.ckook.com)

```
        AlgorithmParameterGenerator apg = AlgorithmParameterGenerator.getInstance
        (KEY_ALGORITHM);
        // 初始化算法参数生成器
        apg.init(KEY_SIZE);
        // 生成算法参数
        AlgorithmParameters params = apg.generateParameters();
        // 构建参数材料
        DHParameterSpec elParams = (DHParameterSpec) params.getParameterSpec
        (DHParameterSpec.class);
        // 实例化密钥对生成器
        KeyPairGenerator kpg = KeyPairGenerator.getInstance(KEY_ALGORITHM);
        // 初始化密钥对生成器
        kpg.initialize(elParams, new SecureRandom());
        // 生成密钥对
        KeyPair keys = kpg.genKeyPair();
        // 取得密钥
        PublicKey publicKey = keys.getPublic();
        PrivateKey privateKey = keys.getPrivate();
        // 封装密钥
        Map<String, Object> map = new HashMap<String, Object>(2);
        map.put(PUBLIC_KEY, publicKey);
        map.put(PRIVATE_KEY, privateKey);
        return map;
    }
    /**
     * 取得私钥
     * @param keyMap 密钥Map
     * @return byte[] 私钥
     * @throws Exception
     */
    public static byte[] getPrivateKey(Map<String, Object> keyMap) throws Exception {
            Key key = (Key) keyMap.get(PRIVATE_KEY);
            return key.getEncoded();
    }
    /**
     * 取得公钥
     * @param keyMap
     * @return
     * @throws Exception
     */
    public static byte[] getPublicKey(Map<String, Object> keyMap)throws Exception {
            Key key = (Key) keyMap.get(PUBLIC_KEY);
            return key.getEncoded();
    }
}
```

ElGamal算法的测试用例与RSA算法测试用例较为相近。不同的是，ElGamal算法实现仅有“公钥加密，私钥解密”的部分。测试用例如代码清单8-13所示。

更多资源请访问稀酷客(www.ckook.com)

代码清单8-13 ElGamal算法实现测试用例

```
import static org.junit.Assert.*;
import java.util.Map;
import org.apache.commons.codec.binary.Base64;
import org.junit.Before;
import org.junit.Test;
/**
 * ElGamal校验
 * @author 梁栋
 * @version 1.0
 */
public class ElGamalCoderTest {
    // 公钥
    private byte[] publicKey;
    // 私钥
    private byte[] privateKey;
    /**
     * 初始化密钥
     * @throws Exception
     */
    @Before
    public void initKey() throws Exception {
        Map<String, Object> keyMap = ElGamalCoder.initKey();
        publicKey = ElGamalCoder.getPublicKey(keyMap);
        privateKey = ElGamalCoder.getPrivateKey(keyMap);
        System.err.println("公钥: \n" + Base64.encodeBase64String(publicKey));
        System.err.println("私钥: \n" + Base64.encodeBase64String(privateKey));
    }
    /**
     * 校验
     * @throws Exception
     */
    @Test
    public void test() throws Exception {
        String inputStr = "ElGamal加密";
        byte[] data = inputStr.getBytes();
        System.err.println("原文: \n" + inputStr);
        byte[] encodedData = ElGamalCoder.encryptByPublicKey(data, publicKey);
        System.err.println("加密后: \n" + Base64.encodeBase64String(encodedData));
        byte[] decodedData = ElGamalCoder.decryptByPrivateKey(encodedData, privateKey);
        String outputStr = new String(decodedData);
        System.err.println("解密后: \n" + outputStr);
        assertEquals(inputStr, outputStr);
    }
}
```

在控制台中的输出信息中，我们可以得到相应的公钥和私钥，如下代码所示：

更多资源请访问稀酷客(www.ckook.com)

```
公钥:
MHcwUAYGKw4HAgEBMEYCIQCutlvZWBGgITJngn6hyMJ/VC/vt7K47W2p7QZdk+xpDwIhAKr3JJo1
jqbZp0YJSeBceSDLL7fJOUATmOzEyXhv0kRcAyMAAiA6HMzcFJSyF78uBXzemyHNFbXOFF0plX15
17p31YQqjQ==
私钥:
MHkCAQAwUAYGKw4HAgEBMEYCIQCutlvZWBGgITJngn6hyMJ/VC/vt7K47W2p7QZdk+xpDwIhAKr3
JJo1jqbZp0YJSeBceSDLL7fJOUATmOzEyXhv0kRcBCICIBecbEByJs28q7NH69zA2xDsjYbx9ihc
IZSZzKO8z/Dn
```

仔细观察，我们发现公钥和私钥的长度几乎是一致的。

观察控制台的输出信息，加密/解密信息如下代码所示：

```
原文:
ElGamal加密
加密后:
T921luoBzFrAkly8I6b9PX9MuuGTiUAcGmn4Zw+iNYozA1BtX/RkhLTtPzDobJQKLAUV3fLN7Jeq
GZsgXC8gOA==
解密后:
ElGamal加密
```

ElGamal算法公钥和私钥长度几乎一致，基于Bouncy Castle加密组件的ElGamal算法实现仅遵循“公钥加密，私钥解密”的简单原则。

8.5 实例：非对称加密网络应用

目前，非对称加密算法（主要是RSA算法）主要应用于B2C、B2B等多种电子商务平台。但非对称加密算法并不直接对网络数据进行加密/解密，而是用于交换对称加密算法的秘密密钥。最终使用对称加密算法进行真正的加密/解密。

此处，我们将对第7章的DataServer应用稍作修改，使用非对称加密算法RSA交换对称加密算法AES的秘密密钥。并使用该秘密密钥对数据进行加密/解密。

对于第7章中用于实现DataServer应用的HttpUtils类和AESCoder类，本文不做详述，请读者阅读相关内容。此处，我们仅对DataServlet类和DataServlet类稍作修改，并增加用于RSA算法实现的RSACoder类。

首先，我们要对本章中的RSACoder类稍作修改：使用开源组件Commons Codec的十六进制转换工具类Hex对密钥进行封装/解包。相关实现如代码清单8-14所示。

代码清单8-14 密钥封装/解包

```
import org.apache.commons.codec.binary.Hex;
// 省略
/**
 * 私钥加密
 * @param data 待加密数据
 * @param key 私钥
 * @return byte[] 加密数据
```

更多资源请访问稀酷客(www.ckook.com)

```
     * @throws Exception
     */
    public static byte[] encryptByPrivateKey(byte[] data, String key) throws Exception {
        return encryptByPrivateKey(data, getKey(key));
    }
    /**
     * 公钥加密
     * @param data 待加密数据
     * @param key 公钥
     * @return byte[] 加密数据
     * @throws Exception
     */
    public static byte[] encryptByPublicKey(byte[] data, String key)
    throws Exception {
        return encryptByPublicKey(data, getKey(key));
    }
    /**
     * 私钥解密
     * @param data 待解密数据
     * @param key 私钥
     * @return byte[] 解密数据
     * @throws Exception
     */
    public static byte[] decryptByPrivateKey(byte[] data, String key)
    throws Exception {
        return decryptByPrivateKey(data, getKey(key));
    }
    /**
     * 公钥解密
     * @param data 待解密数据
     * @param key 私钥
     * @return byte[] 解密数据
     * @throws Exception
     */
    public static byte[] decryptByPublicKey(byte[] data, String key) throws Exception {
        return decryptByPublicKey(data, getKey(key));
    }
    /**
     * 初始化密钥
     * @param keyMap 密钥Map
     * @return String 十六进制编码密钥
     * @throws Exception
     */
    public static String getPrivateKeyString(Map<String, Object> keyMap) throws Exception {
        return Hex.encodeHexString(getPrivateKey(keyMap));
    }
    /**
```

更多资源请访问稀酷客(www.ckook.com)

```
     * 初始化密钥
     * @param keyMap 密钥Map
     * @return String 十六进制编码密钥
     * @throws Exception
     */
    public static String getPublicKeyString(Map<String, Object> keyMap) throws Exception {
        return Hex.encodeHexString(getPublicKey(keyMap));
    }
    /**
     * 获取密钥
     * @param key 密钥
     * @return byte[] 密钥
     * @throws Exception
     */
    public static byte[] getKey(String key) throws Exception {
        return Hex.decodeHex(key.toCharArray());
    }
```

在实际应用中，常常使用Base64或十六进制编码将密钥转为可见字符存储。十六进制编码要比Base64编码长度小得多，读者朋友可以根据需要选择合适的算法。完整实现如代码清单8-15所示。

代码清单8-15 RSACoder

```
import java.security.Key;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.util.HashMap;
import java.util.Map;
import javax.crypto.Cipher;
import org.apache.commons.codec.binary.Hex;
/**
 * RSA安全编码组件
 * @author 梁栋
 * @version 1.0
 */
public abstract class RSACoder {
    // 非对称加密密钥算法
    public static final String KEY_ALGORITHM = "RSA";
    // 公钥
    private static final String PUBLIC_KEY = "RSAPublicKey";
```

更多资源请访问稀酷客(www.ckook.com)

```
// 私钥
private static final String PRIVATE_KEY = "RSAPrivateKey";
// RSA密钥长度默认1024位，密钥长度必须是64的倍数，范围在512～65536位之间。
private static final int KEY_SIZE = 512;
/**
 * 私钥解密
 * @param data 待解密数据
 * @param key 私钥
 * @return byte[] 解密数据
 * @throws Exception
 */
public static byte[] decryptByPrivateKey(byte[] data, byte[] key) throws Exception {
        // 取得私钥
        PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(key);
        KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
        // 生成私钥
        PrivateKey privateKey = keyFactory.generatePrivate(pkcs8KeySpec);
        // 对数据解密
        Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
        cipher.init(Cipher.DECRYPT_MODE, privateKey);
        return cipher.doFinal(data);
}
/**
 * 公钥解密
 * @param data 待解密数据
 * @param key 公钥
 * @return byte[] 解密数据
 * @throws Exception
 */
public static byte[] decryptByPublicKey(byte[] data, byte[] key) throws Exception {
        // 取得公钥
        X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(key);
        KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
        // 生成公钥
        PublicKey publicKey = keyFactory.generatePublic(x509KeySpec);
        // 对数据解密
        Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
        cipher.init(Cipher.DECRYPT_MODE, publicKey);
        return cipher.doFinal(data);
}
/**
 * 公钥加密
 * @param data 待加密数据
 * @param key 公钥
 * @return byte[] 加密数据
 * @throws Exception
 */
public static byte[] encryptByPublicKey(byte[] data, byte[] key) throws Exception {
```

更多资源请访问稀酷客(www.ckook.com)

```
        // 取得公钥
        X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(key);
        KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
        PublicKey publicKey = keyFactory.generatePublic(x509KeySpec);
        // 对数据加密
        Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
        cipher.init(Cipher.ENCRYPT_MODE, publicKey);
        return cipher.doFinal(data);
    }
    /**
     * 私钥加密
     * @param data 待加密数据
     * @param key 私钥
     * @return byte[] 加密数据
     * @throws Exception
     */
    public static byte[] encryptByPrivateKey(byte[] data, byte[] key) throws Exception {
        // 取得私钥
        PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(key);
        KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
        // 生成私钥
        PrivateKey privateKey = keyFactory.generatePrivate(pkcs8KeySpec);
        // 对数据加密
        Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
        cipher.init(Cipher.ENCRYPT_MODE, privateKey);
        return cipher.doFinal(data);
    }
    /**
     * 取得私钥
     * @param keyMap 密钥Map
     * @return byte[] 私钥
     * @throws Exception
     */
    public static byte[] getPrivateKey(Map<String, Object> keyMap) throws Exception {
        Key key = (Key) keyMap.get(PRIVATE_KEY);
        return key.getEncoded();
    }
    /**
     * 取得公钥
     * @param keyMap 密钥Map
     * @return byte[] 公钥
     * @throws Exception
     */
    public static byte[] getPublicKey(Map<String, Object> keyMap) throws Exception {
        Key key = (Key) keyMap.get(PUBLIC_KEY);
        return key.getEncoded();
    }
```

更多资源请访问稀酷客(www.ckook.com)

```
    /**
     * 初始化密钥
     * @return Map 密钥Map
     * @throws Exception
     */
    public static Map<String, Object> initKey() throws Exception {
            // 实例化密钥对生成器
            KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance(KEY_ALGORITHM);
            // 初始化密钥对生成器
            keyPairGen.initialize(KEY_SIZE);
            // 生成密钥对
            KeyPair keyPair = keyPairGen.generateKeyPair();
            // 公钥
            RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();
            // 私钥
            RSAPrivateKey privateKey = (RSAPrivateKey) keyPair.getPrivate();
            // 封装密钥
            Map<String, Object> keyMap = new HashMap<String, Object>(2);
            keyMap.put(PUBLIC_KEY, publicKey);
            keyMap.put(PRIVATE_KEY, privateKey);
            return keyMap;
    }
    /**
     * 私钥加密
     * @param data 待加密数据
     * @param key 私钥
     * @return byte[] 加密数据
     * @throws Exception
     */
    public static byte[] encryptByPrivateKey(byte[] data, String key) throws Exception {
            return encryptByPrivateKey(data, getKey(key));
    }
    /**
     * 公钥加密
     * @param data 待加密数据
     * @param key 公钥
     * @return byte[] 加密数据
     * @throws Exception
     */
    public static byte[] encryptByPublicKey(byte[] data, String key) throws Exception {
            return encryptByPublicKey(data, getKey(key));
    }
    /**
     * 私钥解密
     * @param data 待解密数据
     * @param key 私钥
     * @return byte[] 解密数据
```

更多资源请访问稀酷客(www.ckook.com)

```
     * @throws Exception
     */
    public static byte[] decryptByPrivateKey(byte[] data, String key) throws Exception {
            return decryptByPrivateKey(data, getKey(key));
    }
    /**
     * 公钥解密
     * @param data 待解密数据
     * @param key 私钥
     * @return byte[] 解密数据
     * @throws Exception
     */
    public static byte[] decryptByPublicKey(byte[] data, String key) throws Exception {
            return decryptByPublicKey(data, getKey(key));
    }
    /**
     * 初始化密钥
     * @param keyMap 密钥Map
     * @return String 十六进制编码密钥
     * @throws Exception
     */
    public static String getPrivateKeyString(Map<String, Object> keyMap) throws Exception {
            return Hex.encodeHexString(getPrivateKey(keyMap));
    }
    /**
     * 初始化密钥
     * @param keyMap 密钥Map
     * @return String 十六进制编码密钥
     * @throws Exception
     */
    public static String getPublicKeyString(Map<String, Object> keyMap) throws Exception {
            return Hex.encodeHexString(getPublicKey(keyMap));
    }
    /**
     * 获取密钥
     * @param key 密钥
     * @return byte[] 密钥
     * @throws Exception
     */
    public static byte[] getKey(String key) throws Exception {
            return Hex.decodeHex(key.toCharArray());
    }
}
```

通过调用RSACoder类的相关方法获得公钥和私钥，相关实现如代码清单8-16所示。

代码清单8-16 RSACoder——构建密钥

```
// 初始化密钥
```

更多资源请访问稀酷客(www.ckook.com)

```
Map<String, Object> keyMap = RSACoder.initKey();
// 获得并打印公钥
String publicKey = RSACoder.getPublicKeyString(keyMap);
System.err.println("publicKey - " + publicKey);
// 获得并打印私钥
String privateKey = RSACoder.getPrivateKeyString(keyMap);
System.err.println("privateKey - " + privateKey);
```

我们在控制台得到相应的公钥与私钥信息，我们将在后续实现中直接使用这些密钥。控制台密钥输出如下代码所示：

```
publicKey-
305c300d06092a864886f70d0101010500034b0030480241009fec6cff0209ef1a332a35ccafc2aae59c
4d5275ef9186d73593186482ec637f6df042c2aa41115c8a1625a2e9e9f7844c008389e5a6379a268d87
7fd8edc2690203010001
privateKey-
308201530201003 00d06092a864886f70d01010105000482013d308201390201000241009fec6cff0209
ef1a332a35ccafc2aae59c4d5275ef9186d73593186482ec637f6df042c2aa41115c8a1625a2e9e9f784
4c008389e5a6379a268d877fd8edc269020301000102400275ce73b214256b2e9331388ed1e0a3877ef64
43991305d084e44ed5b68ffeb124274a17a3e3dd6e3013011c0602bb6efebebb5d327ebdf8052766f6d8
be838d022100d372663308ab6e5c057bc5a56f9b9a16602e2872ded9344fa1dfbaadfb38d24b022100c1
9ed1dda201473d8fe6433fbfce1486d18782c837d30d4f122f81157a25ed9b0220047a6bc7b0eb508f0a
5eb0b4ec4433633dee3c55127b2f2c70953872eedb293902204c451bb68a92a65581d1dabbc9fa8beb6f
ae49be44ff4646d78b0ef63edfa1f1022010d9524a9febc784bf8ad8dc07a15dee9f47744f9081599b0c
c5e18fc37cee3f
```

接下来，我们来调整DataSerlvet类。这里，我们使用RSA算法对请求的数据进行解密，获得本次交互的对称加密算法密钥。相关实现如代码清单8-17所示。

代码清单8-17 DataServlet——解析密钥

```
byte[] input = HttpUtils.requestRead(request);
// 对秘密密钥解密
String k = new String(RSACoder.decryptByPrivateKey(input, key));
```

本文演示程序获得密钥后，使用AES算法将数据加密回复给请求方，相关实现如代码清单8-18所示。

代码清单8-18 DataServlet——加密回复

```
// 使用AES算法对数据加密并回复
HttpUtils.responseWrite(response, AESCoder.encrypt(output, k));
```

DataServlet类完整实现如代码清单8-19所示。

代码清单8-19 DataServlet

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
```

更多资源请访问稀酷客(www.ckook.com)

```
import javax.servlet.http.HttpServletResponse;
/**
 * 数据服务DataServlet
 * @author 梁栋
 * @since 1.0
 */
public class DataServlet extends HttpServlet {
    private static final long serialVersionUID = -6219906900195793155L;
    // 密钥
    private static String key;
    // Servlet初始化参数--私钥
    private static final String KEY_PARAM = "key";
    // 初始化
    @Override
    public void init() throws ServletException {
            super.init();
            // 初始化密钥
            key = getInitParameter(KEY_PARAM);
    }
    // 处理POST请求
    protected void doPost(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
            try {
byte[] input = HttpUtils.requestRead(request);
// 对秘密密钥解密
String k = new String(RSACoder.decryptByPrivateKey(input, key));
// 输出秘密密钥
System.err.println(k);
// 构造数据包
StringBuilder sb = new StringBuilder();
sb.append("<?xmi version=\"1.0\"encoding=]"UTF-8\"?>\r\n");
sb.append("<dataGroup>\r\n");
sb.append("\t<dataItem>\r\n");
sb.append("\t\t<id>");
sb.append("10201");
sb.append("</id>\r\n");
sb.append("\t\t<price>");
sb.append("35.0");
sb.append("</price>\r\n");
sb.append("\t\t<time>");
sb.append("2009-10-30");
sb.append("</time>\r\n");
sb.append("\t</dataItem>\r\n");
sb.append("\t<dataItem>\r\n");
sb.append("\t\t<id>");
sb.append("10301");
sb.append("</id>\r\n");
sb.append("\t\t<price>");
```

更多资源请访问稀酷客(www.ckook.com)

```
                    sb.append("55.0");
                    sb.append("</price>\r\n");
                    sb.append("\t\t<time>");
                    sb.append("2009-10-31");
                    sb.append("</time>\r\n");
                    sb.append("\t</dataItem>\r\n");
                    sb.append("</dataGroup>\r\n");
                    byte[] output = sb.toString().getBytes();
                    // 使用AES算法对数据加密并回复
                    HttpUtils.responseWrite(response, AESCoder.encrypt(output, k));
                    } catch (Exception e) {
                            throw new ServletException(e);
                    }
    }
}
```

同时，我们需要修改web.xml文件，将私钥作为密钥变量配置其中。完整实现如代码清单8-20所示。

代码清单8-20　web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
    app_2_5.xsd"
    id="WebApp_ID"
    version="2.5">
    <display-name>DataServer</display-name>
    <servlet>
        <servlet-name>DataServlet</servlet-name>
        <servlet-class>DataServlet</servlet-class>
        <init-param>
            <param-name>key</param-name>
            <param-value>
  <![CDATA[308201530201003000d06092a864886f70d01010105000482013d308201390201000241009fec6cff0209ef1a332a35ccafc2aae59c4d5275ef9186d73593186482ec637f6df042c2aa41115c8a1625a2e9e9f7844c008389e5a6379a268d877fd8edc2690203010001024075ce73b214256b2e9331388ed1e0a3877ef6443991305d084e44ed5b68ffeb124274a17a3e3dd6e3013011c0602bb6efebebb5d327ebdf8052766f6d8be838d022100d372663308ab6e5c057bc5a56f9b9a16602e2872ded9344fa1dfbaadfb38d24b022100c19ed1dda201473d8fe6433fbfce1486d18782c837d30d4f122f81157a25ed9b0220047a6bc7b0eb508f0a5eb0b4ec4433633dee3c55127b2f2c70953872eedb293902204c451bb68a92a65581d1dabbc9fa8beb6fae49be44ff4646d78b0ef63edfa1f1022010d9524a9febc784bf8ad8dc07a15dee9f47744f9081599b0cc5e18fc37cee3f]]>
            </param-value>
        </init-param>
    </servlet>
```

更多资源请访问稀酷客(www.ckook.com)

```
        <servlet-mapping>
            <servlet-name>DataServlet</servlet-name>
            <url-pattern>/DataServlet</url-pattern>
        </servlet-mapping>
    </web-app>
```

最后，我们来调整测试用例DataServletTest类。

此处，我们先通过AESCoder类initKeyString()方法构建秘密密钥，使用RSACoder对其加密并将其发送给服务器。相关实现如代码清单8-21所示。

代码清单8-21 DataServletTest——构建秘密密钥

```
    /**
     * 公钥
     */
    private static final String publicKey =
    "305c300d06092a864886f70d0101010500034b0030480241009fec6cff0209ef1a332a35ccafc2a
ae59c4d5275ef9186d73593186482ec637f6df042c2aa41115c8a1625a2e9e9f7844c008389e5a6379a2
68d877fd8edc2690203010001";
    // 省略
    // 构建秘密密钥
    String secretKey = AESCoder.initKeyString();
    // 使用RSA算法加密并发送秘密密钥
    byte[] input = HttpUtils.postRequest(url,
    RSACoder.encryptByPublicKey(secretKey.getBytes(), publicKey));
```

最后，我们使用AES算法对收到的数据进行解密，如代码清单8-22所示。

代码清单8-22 DataServletTest——数据解密

```
    // 使用AES算法对数据解密
    String data = new String(AESCoder.decrypt(input, secretKey));
```

完整实现如代码清单8-23所示。

代码清单8-23 DataServletTest

```
    import static org.junit.Assert.*;
    import org.junit.Test;
    /**
     * DataServlet测试用例
     * @author 梁栋
     * @since 1.0
     */
    public class DataServletTest {
        // 公钥
        private static final String publicKey =
    "305c300d06092a864886f70d0101010500034b0030480241009fec6cff0209ef1a332a35ccafc2a
ae59c4d5275ef9186d73593186482ec637f6df042c2aa41115c8a1625a2e9e9f7844c008389e5a6379a2
68d877fd8edc2690203010001";
```

更多资源请访问稀酷客(www.ckook.com)

```
    // 请求地址
    private static final String url = "http://localhost:8080/dataserver/DataServlet";
    @Test
    public final void test() throws Exception {
            // 构建秘密密钥
            String secretKey = AESCoder.initKeyString();
            // 使用RSA算法加密并发送秘密密钥
            byte[] input = HttpUtils.postRequest(url, RSACoder.encryptByPublicKey
            (secretKey.getBytes(), publicKey));
            // 使用AES算法对数据解密
            String data = new String(AESCoder.decrypt(input, secretKey));
            System.err.println(data);
            // 校验
            assertNotNull(data);
    }
}
```

启动DataServer服务，执行DataServletTest测试方法。

最终，我们可以在控制台得到由服务器下发的数据信息，如以下代码所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<dataGroup>
    <dataItem>
            <id>10201</id>
            <price>35.0</price>
            <time>2009-10-30</time>
    </dataItem>
    <dataItem>
            <id>10301</id>
            <price>55.0</price>
            <time>2009-10-31</time>
    </dataItem>
</dataGroup>
```

在实际应用中，我们很少会直接使用非对称加密算法进行数据加密。真正对数据进行加密的算法其实都是对称加密算法。非对称加密算法的主要职责是用来初始化对称加密算法的秘密密钥。

这里，对比第7章代码实现我们缺少了对于数据校验的部分。对于这项关键操作我们有两种选择：

1）使用消息摘要算法对其数据进行摘要/验证。

2）使用数字签名对其数据进行签名/验证。

有关数字签名，以及对上述代码的改进我们将在第9章中详述。

非对称加密算法通常配合数字证书、SSL/TLS协议构建单向认证或双向认证使用，请读者朋友关注后续章节内容。

更多资源请访问稀酷客(www.ckook.com)

8.6 小结

对称加密算法提高数据安全性的同时带来了密钥管理的复杂性。消息收发双方若想发送加密消息，必须事先约定好加密算法并发放密钥。加密消息在传递过程中，难免会被窃听。如果窃听者破译了密钥，就可以破译、甚至篡改消息，而消息的收发双方却浑然不知。当然，我们可以通过消息摘要算法验证消息的完整性，但却不能避免消息被破译的可能。消息收发双方若要避免这种密钥被破解的风险，就必须频繁更换密钥，而密钥的发放又是一个敏感问题，操作起来具有一定的难度。

非对称加密算法构建了两把密钥：私钥保密，称为私钥；公钥公开，称为公钥。公钥与私钥一一对应，且不能相互推导得出，即便公钥在传输过程中被截获也不能推导出私钥。也就是说，即便公钥被截获或被破解，也无法推到出对应的私钥。非对称加密算法普遍遵从“公钥加密，私钥解密”和“私钥加密，公钥解密”这两种加密方式。

在非对称加密算法的发展史上主要包括DH、RSA、ElGamal和ECC共4种算法。

DH（Diffie-Hellman，密钥交换算法）算法由W.Diffie和M.Hellman提出，构建了最原始的非对称加密算法实现方式，为后续非对称加密算法实现奠定了基础。交互双方通过交换秘密密钥，使用对称加密算法完成加密消息传递。Java 6对DH算法提供了完善的支持，并将DH算法实现作为后续非对称加密算法实现的基础。

继DH算法提出后，国际上相继提出了两种基于数学问题求解的非对称加密算法：基于因子分解难题和基于离散对数难题。

基于因子分解难题的非对称加密算法目前仅有RSA算法一种。基于离散对数难题的非对称加密算法包括ElGamal和ECC算法。

RSA算法由美国麻省理工学院（MIT）的Ron Rivest，AdiShamir 和Leonard Adleman三位学者提出，并以三位学者的姓氏开头字母命名。RSA算法是唯一被广泛接受并实现的通用公开加密算法，目前已经成为非对称加密算法的国际标准。不仅如此，RSA算法既可用于数据加密也可用于数字签名。我们熟知的电子邮件加密软件PGP就采用了RSA算法对数据进行加密/解密和数字签名处理。Java 6提供了完善的RSA算法支持，通过Bouncy Castle可以扩充相应的填充方式。

Taher ElGamal提出了ElGamal算法，并以自己的名字命名。该算法即可用于加密/解密，也可用于数字签名，并为数字签名算法形成标准提供参考。美国的DSS（Digital Signature Standard，数据签名标准）的DSA（Digital Signature Algorithm，数字签名算法）经ElGamal算法演变而来。Java 6本身没有提供ElGamal算法支持，我们可以通过Bouncy Castle扩充对概算法的支持。

ECC（Elliptical Curve Cryptography，椭圆曲线加密）算法以椭圆曲线理论为基础，在创建密钥时可做到更快、更小，并且更有效。ECC 算法通过椭圆曲线方程式的性质产生密钥，而不是采用传统的方法利用大质数的积来产生。目前，对于该算法尚无基于Java语言的开源组件。

非对称加密算法的出现并不意味着对称加密算法要退出历史的舞台。非对称加密算法的安全性较高，但却无法回避加密/解密低效问题。

目前，非对称加密算法多应用于安全性要求较高的场合，如B2B、B2C等电子商务领域。

更多资源请访问稀酷客(www.ckook.com)

第9章

带密钥的消息摘要算法——数字签名算法

相信绝大多数读者朋友都有刷卡购物的经历。刷卡时，我们都要签下自己的名字，我们把这种行为简称为“签名”，或者叫做“手写签名”。这种签名几乎与使用者一一绑定，成为防止使用者对其行为否认的一种手段。如果使用者对自己的行为表示质疑，可鉴定签名是否一致，判别签名是否有效。

数字签名是手写签名在计算机软件应用中的一种体现，它同样起到了抗否认的作用。手写签名应用于纸质文件，数字签名应用于数据。手写签名针对纸质文件内容确认，数字签名对数据进行摘要处理。两种方式都离不开签名实体，其校验方法也基本一致。

无论是手写签名还是数字签名，如果离开了签名实体（文件或数据）就没有了意义。如果经过手写签名的文件被修改了，我们可以认为该文件无效。同理，经过数字签名的数据可以通过验证签名操作辨别该数据是否被修改。

相信大多数读者朋友都使用过微软的操作系统（如Windows XP），在安装该系统时需要输入一个25位的产品密钥，系统会验证这个产品密钥是否合法，这其实就是一个签名验证的过程。

9.1 数字签名算法简述

数字签名算法可以看做是一种带有密钥的消息摘要算法，并且这种密钥包含了公钥和私钥。也就是说，数字签名算法是非对称加密算法和消息摘要算法的结合体。

9.1.1 数字签名算法的由来

数字签名算法是公钥基础设施（Public Key Infrastructure，PKI）以及许多网络安全机制（SSL/TLS、VPN等）的基础。

数字签名算法要求能够验证数据完整性、认证数据来源，并起到抗否认的作用。这3点与OSI参考模型中的数据完整性服务、认证（鉴别）服务和抗否认性服务相对应。

消息摘要算法是验证数据完整性的最佳算法，因此，该算法成为数字签名算法中的必要组成部分。

基于数据完整性验证，我们希望数据的发送方（以下称甲方）可以对自己所发送的数据做

更多资源请访问稀酷客(www.ckook.com)

相应的处理（签名处理），同时给出对应的凭证（签名），并且数据的接收方（以下称乙方）可以验证该签名是否与数据甲方发送的数据相符。

如果任何机构都可以进行签名处理，那签名本身就失去了验证的意义。因此，签名操作只能由甲方来完成，验证签名操作则由乙方来完成。既然签名操作仅限于甲方，那么签名操作本身是基于甲方的某些私有信息完成的操作。并且，用于验证操作的相关信息是由甲方公布给乙方。

用于签名的相关信息私有，用于验证的相关信息公有，且这两种信息必须成对出现。非对称加密算法中的私钥和公钥满足这种关系，成为数字签名算法中的重要元素。

数字签名算法包含签名和验证两项操作，遵循“私钥签名，公钥验证”的签名/验证方式，签名时需要使用私钥和待签名数据，验证时则需要公钥、签名值和待签名数据，其核心算法主要是消息摘要算法。因此，我们可以把数字签名算法近似看成是一种附加了公钥和私钥的消息摘要算法。

与摘要值的表示方式相同，签名值也常以十六进制字符串的形式来表示。

需要读者朋友注意的是，数字签名算法在实际运用时，通常是先使用消息摘要算法对原始消息做摘要处理，然后再使用私钥对摘要值做签名处理；验证签名时，则使用公钥验证消息的摘要值。

9.1.2 数字签名算法的家谱

数字签名算法主要包括RSA、DSA和ECDSA共3种算法。其中，RSA算法源于整数因子分解问题，DSA和ECDSA算法源于离散对数问题。

作为非对称加密算法，RSA算法堪称典型，同样也是数字签名算法中的经典。基于RSA算法密钥，结合消息摘要算法可形成对应的签名算法。如结合消息摘要算法MD5算法，可形成MD5withRSA算法。

DSA算法是继RSA算法后出现的基于DSS的数字签名算法，旨在形成数字签名标准。DSA算法主要为后续数字签名算法的形成奠定基础。

ECDSA算法是椭圆曲线加密算法ECC与DSA算法的结合，相对于传统签名算法，它具有速度快、强度高、签名短等优点，其用途也越来越广泛。

上述算法中，在Java语言中都可获得相应的支持。其中，Java 6完全实现了DSA算法，部分RSA算法需要由Bouncy Castle提供支持，ECDSA算法则完全需要Bouncy Castle提供支持。

9.2 模型分析

我们继续以消息传递模型为例，介绍基于数字签名算法的消息传递模型。

数字签名算法在应用领域的使用较为简单，在密钥处理方面与一般非对称加密算法无异，只是将加密/解密换成了签名/验证。

无论我们将要介绍哪一种数字加密算法，在构建密钥对这一操作中，都与非对称加密算法无异，尤其是与RSA算法完全一致。读者朋友可阅读本书第8章相关内容。

更多资源请访问稀酷客(www.ckook.com)

在图9-1中，甲方作为消息的发送方，乙方作为消息的接收方。我们假设甲乙双方在消息传递之前已将指定了将要使用的数字签名算法（如RSA算法）。为完成签名验证，甲乙双方需要以下操作：

1）由消息发送的一方构建密钥对，这里由甲方完成。

2）由消息发送的一方公布公钥至消息接收方，这里由甲方将公钥公布给乙方。

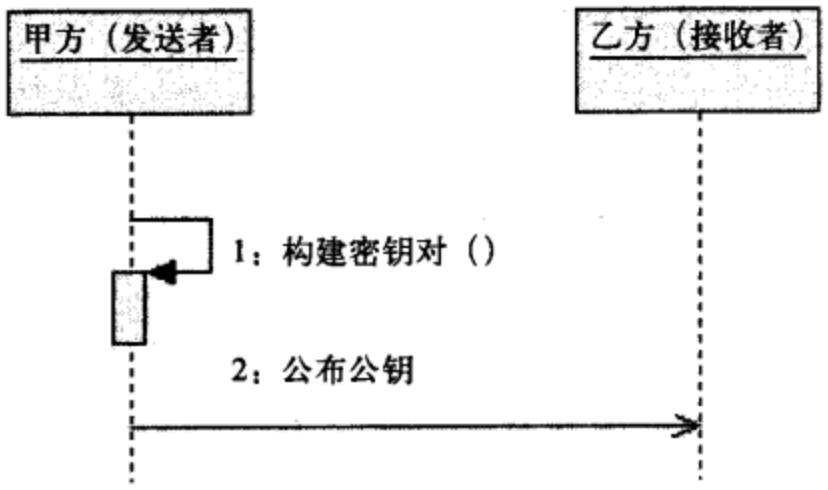


图9-1　构建密钥对

完成这两步操作后，甲方向乙方发送的数据就可以做验证了，如图9-2所示。

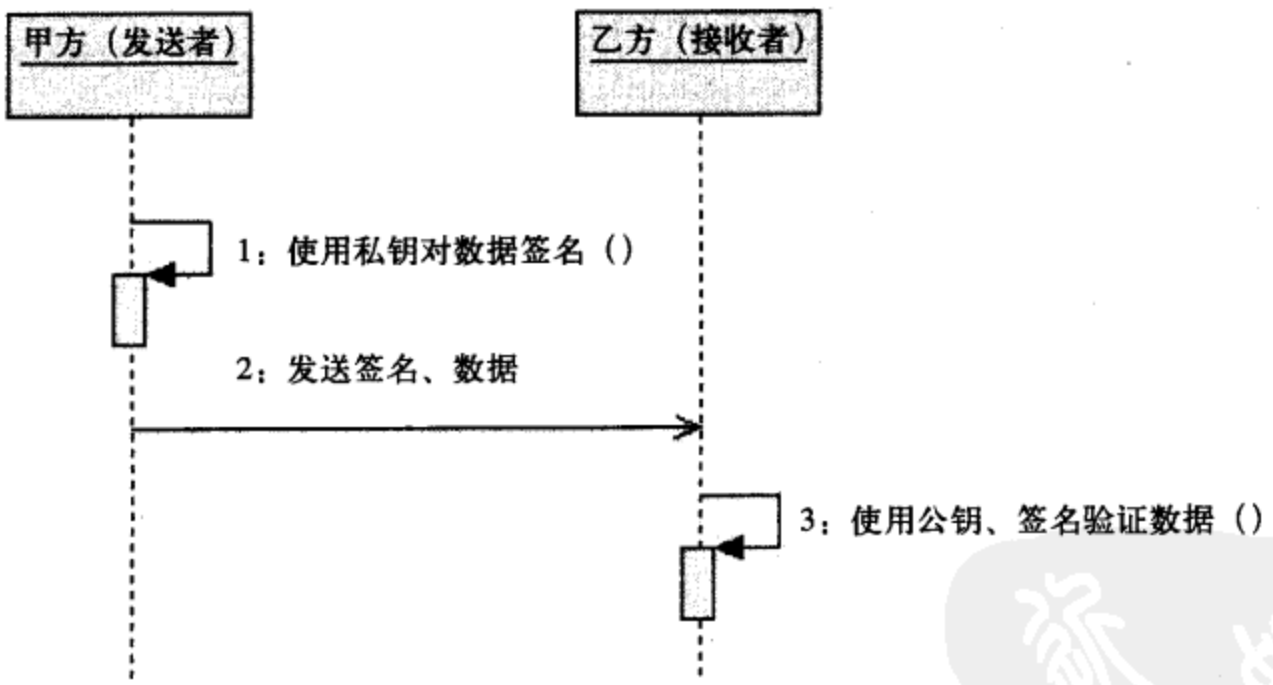


图9-2　甲方向乙方发送数据

在图9-2中，甲方向乙方发送数据时需要附加签名，数据与签名形成一则消息发送给接收者。如本文开篇所讲，签名与实体（这里指签名前的数据）不可分离，作为一个整体发送给乙方。并且，私钥仅用于签名，公钥仅用于验证。

9.3　经典数字签名算法——RSA

RSA算法既是非对称加密算法中的经典，同样也是数字签名算法中的经典。而且，在Java语言的世界里，对于RSA算法的支持是最为完整的。

更多资源请访问稀酷客(www.ckook.com)

9.3.1 简述

RSA数字签名算法是Diffie和Hellman提出数字签名思想后的第一个数字签名算法，是由Rivest、Shamir和Adleman三人共同完成的，该签名算法源于RSA公钥密码算法的思想，将RSA公钥密码算法按照数字签名的方式运用。RSA数字签名算法是迄今为止应用最为广泛的数字签名算法。

9.3.2 实现

Java 6不仅提供了RSA加密算法相关实现，同时也提供了RSA数字签名算法实现。

RSA数字签名算法的密钥实现与RSA加密算法一致，算法名称同为“RSA”，密钥产生与转换完全一致，读者朋友可阅读第8章相关内容。

RSA数字签名算法主要可以分为MD系列和SHA系列两大类。MD系列主要包括MD2withRSA和MD5withRSA共2种数字签名算法；SHA系列主要包括SHA1withRSA、SHA224withRSA、SHA256withRSA、SHA384withRSA和SHA512withRSA共5种数字签名算法。其中，SHA224withRSA、SHA256withRSA、SHA384withRSA和SHA512withRSA这4种数字签名算法需要由第三方加密组件包提供，例如Bouncy Castle。Java 6则只提供了MD2withRSA、MD5withRSA和SHA1withRSA共3种数字签名算法。

有关RSA数字签名算法的Java 6实现与Bouncy Castle实现细节如表9-1所示。

表9-1 RSA数字签名算法

算法	密钥长度	密钥长度默认值	签名长度	备注
MD2withRSA	512～65536位（密钥长度必须是64的倍数）	1024	与密钥长度相同	Java 6实现
MD5withRSA				
SHA1withRSA				
SHA224withRSA		2048		Bouncy Castle实现
SHA256withRSA				
SHA384withRSA				
SHA512withRSA				
RIPEMD128withRSA				
RIPEMD160withRSA				

经作者反复测试，RSA数字签名算法的签名值与密钥长度相同。

我们只需要注意签名和验证两个方法的实现，如代码清单9-1和9-2所示。相关Java API内容请阅读第3章。

代码清单9-1 RSA数字签名算法——签名

```
// 实例化Signature
Signature signature = Signature.getInstance("MD5withRSA");
// 初始化Signature
```

更多资源请访问稀酷客(www.ckook.com)

```
signature.initSign(privateKey);
// 更新
signature.update(data);
// 签名
byte[] sign = signature.sign();
```

注意：上述代码清单中使用到的变量privateKey是私钥对象，data则是待签名数据，sign就是我们需要的签名。

代码清单9-2 RSA数字签名算法——验证

```
// 实例化Signature
Signature signature = Signature.getInstance("MD5withRSA");
// 初始化Signature
signature.initVerify(publicKey);
// 更新
signature.update(data);
// 验证
boolean status = signature.verify(sign);
```

注意上述代码，验证签名时使用到的变量publicKey是公钥对象，data则是待验证数据，sign就是我们之前得到的签名。

数字签名与验证实现如上述代码清单9-1和9-2所示，较为简单。

需要注意的是， RSA密钥长度默认1024位，密钥长度必须是64的倍数，范围在512～65536位之间。

我们以RSA数字签名算法MD5withRSA为例，展示完整的数字签名算法实现，如代码清单9-3所示。

代码清单9-3 RSA数字签名算法实现

```
import java.security.Key;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.Signature;
import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.util.HashMap;
import java.util.Map;
/**
 * RSA签名算法组件
 * @author 梁栋
 * @version 1.0
 */
```

更多资源请访问稀酷客(www.ckook.com)

```
public abstract class RSACoder {
    /**
     * 数字签名
     * 密钥算法
     */
    public static final String KEY_ALGORITHM = "RSA";
    /**
     * 数字签名
     * 签名/验证算法
     */
    public static final String SIGNATURE_ALGORITHM = "MD5withRSA";
    // 公钥
    private static final String PUBLIC_KEY = "RSAPublicKey";
    // 私钥
    private static final String PRIVATE_KEY = "RSAPrivateKey";
    /**
     * RSA密钥长度 默认1024位,
     * 密钥长度必须是64的倍数,
     * 范围在512~65536位之间。
     */
    private static final int KEY_SIZE = 512;
    /**
     * 签名
     * @param data 待签名数据
     * @param privateKey 私钥
     * @return byte[] 数字签名
     * @throws Exception
     */
    public static byte[] sign(byte[] data, byte[] privateKey) throws Exception {
        // 转换私钥材料
        PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(privateKey);
        // 实例化密钥工厂
        KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
        // 取私钥匙对象
        PrivateKey priKey = keyFactory.generatePrivate(pkcs8KeySpec);
        // 实例化Signature
        Signature signature = Signature.getInstance(SIGNATURE_ALGORITHM);
        // 初始化Signature
        signature.initSign(priKey);
        // 更新
        signature.update(data);
        // 签名
        return signature.sign();
    }
    /**
     * 校验
     * @param data 待校验数据
     * @param publicKey 公钥
```

更多资源请访问稀酷客(www.ckook.com)

```
     * @param sign 数字签名
     * @return boolean 校验成功返回true 失败返回false
     * @throws Exception
     */
    public static boolean verify(byte[] data, byte[] publicKey, byte[] sign) throws
    Exception {
            // 转换公钥材料
            X509EncodedKeySpec keySpec = new X509EncodedKeySpec(publicKey);
            // 实例化密钥工厂
            KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
            // 生成公钥
            PublicKey pubKey = keyFactory.generatePublic(keySpec);
            // 实例化Signature
            Signature signature = Signature.getInstance(SIGNATURE_ALGORITHM);
            // 初始化Signature
            signature.initVerify(pubKey);
            // 更新
            signature.update(data);
            // 验证
            return signature.verify(sign);
    }
    /**
     * 取得私钥
     * @param keyMap 密钥Map
     * @return byte[] 私钥
     * @throws Exception
     */
    public static byte[] getPrivateKey(Map<String, Object> keyMap) throws Exception {
            Key key = (Key) keyMap.get(PRIVATE_KEY);
            return key.getEncoded();
    }
    /**
     * 取得公钥
     * @param keyMap 密钥Map
     * @return byte[] 公钥
     * @throws Exception
     */
    public static byte[] getPublicKey(Map<String, Object> keyMap) throws Exception {
            Key key = (Key) keyMap.get(PUBLIC_KEY);
            return key.getEncoded();
    }
    /**
     * 初始化密钥
     * @return Map 密钥Map
     * @throws Exception
     */
    public static Map<String, Object> initKey() throws Exception {
            // 实例化密钥对生成器
```

更多资源请访问稀酷客(www.ckook.com)

```
            KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance(KEY_ALGORITHM);
            // 初始化密钥对生成器
            keyPairGen.initialize(KEY_SIZE);
            // 生成密钥对
            KeyPair keyPair = keyPairGen.generateKeyPair();
            // 公钥
            RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();
            // 私钥
            RSAPrivateKey privateKey = (RSAPrivateKey) keyPair.getPrivate();
            // 封装密钥
            Map<String, Object> keyMap = new HashMap<String, Object>(2);
            keyMap.put(PUBLIC_KEY, publicKey);
            keyMap.put(PRIVATE_KEY, privateKey);
            return keyMap;
    }
}
```

在这里，我们用十六进制字符串形式来表示数字签名，密钥仍以Base64编码表示，上述代码对应的测试用例如代码清单9-4所示。

代码清单9-4 RSA数字签名算法实现测试用例

```
import static org.junit.Assert.*;
import org.apache.commons.codec.binary.Base64;
import org.apache.commons.codec.binary.Hex;
import org.junit.Before;
import org.junit.Test;
import java.util.Map;
/**
 * RSA数字签名校验
 * @author 梁栋
 * @version 1.0
 */
public class RSACoderTest {
    // 公钥
    private byte[] publicKey;
    // 私钥
    private byte[] privateKey;
    /**
     * 初始化密钥
     * @throws Exception
     */
    @Before
    public void initKey() throws Exception {
            Map<String, Object> keyMap = RSACoder.initKey();
            publicKey = RSACoder.getPublicKey(keyMap);
            privateKey = RSACoder.getPrivateKey(keyMap);
            System.err.println("公钥: \n" + Base64.encodeBase64String(publicKey));
```

更多资源请访问稀酷客(www.ckook.com)

```
            System.err.println("私钥: \n" + Base64.encodeBase64String(privateKey));
    }
    /**
     * 校验
     * @throws Exception
     */
    @Test
    public void testSign() throws Exception {
            String inputStr = "RSA数字签名";
            byte[] data = inputStr.getBytes();
            // 产生签名
            byte[] sign = RSACoder.sign(data, privateKey);
            System.err.println("签名:\n" + Hex.encodeHexString(sign));
            // 验证签名
            boolean status = RSACoder.verify(data, publicKey, sign);
            System.err.println("状态:\n" + status);
            // 验证
            assertTrue(status);
    }
}
```

控制台中得到的密钥信息如下所示：

```
公钥:
MFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAKW1jHvKUUwN3zXAtooVJMRn57viQPspDgnluB7Madjd
n++vb5MfyNSc4gIMBKT7t+5H1hPEhHSoDCIDWYHuoFUCAwEAAQ==
私钥:
MIIBUwIBADANBgkqhkiG9w0BAQEFAASCAT0wggE5AgEAAkEApbWMe8pRTA3fNcC2ihUkxGfnu+JA
+ykOCeW4Hsxp2N2f769vkx/I1JziAgwEpPu37kfWE8SEdKgMIgNZge6gVQIDAQABAkAc4/QkOPfH
jLuXwYuRs3H/lCYAyceOgm/iJdzd8cGaf3DO1r65fikzMJtchsUfXCkqUHawDhOezb6BRcAoWVzZ
AiEA6qcVL/V43ugHDcw5UViI0/7o0hCe/FoOTc7OxMZUi/8CIQC0yNBvBQJPkWatJ7/uzbJ2z9VK
tgLmlrGkiVSLGK7jqwIgUkWWtiP45x6vKnVKO20xPDMJ6m0NcSkaDsnN0UxAZH0CIAUUx9b2+xOl
cI9ZWKmrmmKAgOwypW45uhB306p9LR01AiABytSbhV3FsoR+Ypc8H7vMLhvqhHM1gt2GoGlFCdbc
pA==
```

毋庸置疑，对于RSA算法密钥，公钥通常要比私钥短。

注意控制台输出的签名及验证结果，如下所示：

```
签名:
6246e5efec4aa3dd054f846695ac6549e4a90c4e3625a11127d836ce56671fb32f100f7c8a662a38
53ff2e7a1d25c49dd85e67fc240c7fab2cd0ddc422acf55a
状态:
true
```

这个签名串是一个128位的十六进制串，也就是512位的二进制串。在上述测试用例中，我们定义的密钥长度是512位，与签名长度相符。

更多资源请访问稀酷客(www.ckook.com)

9.4 数字签名标准算法——DSA

RSA作为经典数字签名算法，很快就成了数字签名算法的研究对象，并逐步转为标准——DSS，并形成了DSA算法，这为后续数字签名算法的提出奠定了基础，如ECDSA（椭圆曲线数字签名算法）。

9.4.1 简述

1991年，美国国家标准技术协会公布了数字签名标准（Digital Signature Standard，DSS），于1994年正式生效，并作为美国联邦信息处理标准。DSS本质上是ElGamal数字签名算法，DSS使用的算法称为数字签名算法（Digital Signature Algorithm，DSA）。

DSA算法与RSA算法都是数字证书中不可或缺的两种算法。两者不同的是，DSA算法仅包含数字签名算法，使用DSA算法的数字证书无法进行加密通信，而RSA算法既包含加密/解密算法，同时兼有数字签名算法。

9.4.2 实现

Java 6提供了DSA算法实现，在实现层面，我们可以认为DSA算法实现就是RSA数字签名算法实现的简装版。与RSA数字签名算法实现相比，DSA算法仅支持SHA系列的消息摘要算法。Java 6仅支持SHA1withDSA算法，通过Bouncy Castle可以扩展SHA224withDSA、SHA256withDSA、SHA384withDSA和SHA512withDSA共4种数字签名算法。

有关DSA算法的Java 6实现和Bouncy Castle实现如表9-2所示。

表9-2 DSA算法

算法	密钥长度	密钥长度默认值	签名长度	备注
SHA1withDSA	512～1024位（密钥长度必须是64的倍数）	1024	–	Java 6实现
SHA224withDSA				Bouncy Castle实现
SHA256withDSA				
SHA384withDSA				
SHA512withDSA				

需要注意的是，DSA密钥长度默认为1024位，密钥长度必须是64的倍数，范围在512～1024位之间（含）。DSA算法的签名长度与密钥长度无关，且长度不唯一。

具体实现如代码清单9-5所示。

代码清单9-5 DSA算法实现

```
import java.security.Key;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
```

更多资源请访问稀酷客(www.ckook.com)

```
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.SecureRandom;
import java.security.Signature;
import java.security.interfaces.DSAPrivateKey;
import java.security.interfaces.DSAPublicKey;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.util.HashMap;
import java.util.Map;
/**
 * DSA安全编码组件
 * @author 梁栋
 * @version 1.0
 */
public abstract class DSACoder {
    // 数字签名密钥算法
    public static final String ALGORITHM = "DSA";
    /**
     * 数字签名
     * 签名/验证算法
     */
    public static final String SIGNATURE_ALGORITHM = "SHA1withDSA";
    // 公钥
    private static final String PUBLIC_KEY = "DSAPublicKey";
    // 私钥
private static final String PRIVATE_KEY = "DSAPrivateKey";
/**
 * DSA密钥长度
 * 默认1024位
 * 密钥长度必须是64的倍数
 * 范围在512～1024位之间 (含)
 */
private static final int KEY_SIZE = 1024;
/**
 * 签名
 * @param data 待签名数据
 * @param privateKey 私钥
 * @return byte[] 数字签名
 * @throws Exception
 */
public static byte[] sign(byte[] data, byte[] privateKey) throws Exception {
        // 还原私钥
        // 转换私钥材料
        PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(privateKey);
        // 实例化密钥工厂
        KeyFactory keyFactory = KeyFactory.getInstance(ALGORITHM);
        // 生成私钥对象
```

更多资源请访问稀酷客(www.ckook.com)

```
        PrivateKey priKey = keyFactory.generatePrivate(pkcs8KeySpec);
        // 实例化Signature
        Signature signature = Signature.getInstance(SIGNATURE_ALGORITHM);
        // 初始化Signature
        signature.initSign(priKey);
        // 更新
        signature.update(data);
        // 签名
        return signature.sign();
    }
    /**
     * 校验
     * @param data 待校验数据
     * @param publicKey 公钥
     * @param sign 数字签名
     * @return boolean 校验成功返回true 失败返回false
     * @throws Exception
     */
    public static boolean verify(byte[] data, byte[] publicKey, byte[] sign)
    throws Exception {
        // 还原公钥
        // 转换公钥材料
        X509EncodedKeySpec keySpec = new X509EncodedKeySpec(publicKey);
        // 实例化密钥工厂
        KeyFactory keyFactory = KeyFactory.getInstance(ALGORITHM);
        // 取公钥匙对象
        PublicKey pubKey = keyFactory.generatePublic(keySpec);
        // 实例话Signature
        Signature signature = Signature.getInstance(SIGNATURE_ALGORITHM);
        // 初始化Signature
        signature.initVerify(pubKey);
        // 更新
        signature.update(data);
        // 验证
        return signature.verify(sign);
    }
    /**
     * 生成密钥
     * @return 密钥对象
     * @throws Exception
     */
    public static Map<String, Object> initKey() throws Exception {
        // 初始化密钥对生成器
        KeyPairGenerator keygen = KeyPairGenerator.getInstance(ALGORITHM);
        // 实例化密钥对生成器
        keygen.initialize(KEY_SIZE, new SecureRandom());
        // 实例化密钥对
```

更多资源请访问稀酷客(www.ckook.com)

```
        KeyPair keys = keygen.genKeyPair();
        DSAPublicKey publicKey = (DSAPublicKey) keys.getPublic();
        DSAPrivateKey privateKey = (DSAPrivateKey) keys.getPrivate();
        // 封装密钥
        Map<String, Object> map = new HashMap<String, Object>(2);
        map.put(PUBLIC_KEY, publicKey);
        map.put(PRIVATE_KEY, privateKey);
        return map;
    }
    /**
     * 取得私钥
     * @param keyMap 密钥Map
     * @return byte[] 私钥
     * @throws Exception
     */
    public static byte[] getPrivateKey(Map<String, Object> keyMap) throws Exception {
        Key key = (Key) keyMap.get(PRIVATE_KEY);
        return key.getEncoded();
    }
    /**
     * 取得公钥
     * @param keyMap 密钥Map
     * @return byte[] 公钥
     * @throws Exception
     */
    public static byte[] getPublicKey(Map<String, Object> keyMap) throws Exception {
        Key key = (Key) keyMap.get(PUBLIC_KEY);
        return key.getEncoded();
    }
}
```

DSA算法实现是最为简单的数字签名算法，若仅仅需要使用数字签名算法，则DSA算法是不错的选择。当然，DSA算法使用起来就更为简单了，测试用例如代码清单9-6所示。

代码清单9-6 DSA算法实现测试用例

```
import static org.junit.Assert.*;
import java.util.Map;
import org.apache.commons.codec.binary.Base64;
import org.apache.commons.codec.binary.Hex;
import org.junit.Before;
import org.junit.Test;
/**
 * DSA签名校验
 * @author 梁栋
 * @version 1.0
 */
public class DSACoderTest {
```

更多资源请访问稀酷客(www.ckook.com)

```
    // 公钥
    private byte[] publicKey;
    // 私钥
    private byte[] privateKey;
    /**
     * 初始化密钥
     * @throws Exception
     */
    @Before
    public void initKey() throws Exception {
            Map<String, Object> keyMap = DSACoder.initKey();
            publicKey = DSACoder.getPublicKey(keyMap);
            privateKey = DSACoder.getPrivateKey(keyMap);
            System.err.println("公钥: \n" + Base64.encodeBase64String(publicKey));
            System.err.println("私钥:  \n" + Base64.encodeBase64String(privateKey));
    }
    /**
     * 校验签名
     * @throws Exception
     */
    @Test
    public void test() throws Exception {
            String inputStr = "DSA数字签名";
            byte[] data = inputStr.getBytes();
            // 产生签名
            byte[] sign = DSACoder.sign(data, privateKey);
            System.err.println("签名:\r" + Hex.encodeHexString(sign));
            // 验证签名
            boolean status = DSACoder.verify(data, publicKey, sign);
            System.err.println("状态:\r" + status);
            assertTrue(status);
    }
}
```

DSA算法的公钥长度略长于私钥，如下所示：

```
公钥:
MIIBuDCCASwGByqGSM44BAEwggEfAoGBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2USZp
RV1AIlH7WT2NWPq/xfW6MPbLm1Vs14E7gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1fn
xqimFQ8E+4P208UewwI1VBNaFpEy9nXzrith1yrv8iIDGZ3RSAHHAhUAl2BQjxUjC8yykrmCouuE
C/BYHPUCgYEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCBgLRJ
FnEj6EwoFhO3zwkyjMim4TwWeotUfI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImo
g9/hWuWfBpKLZl6Ae1UlZAFMO/7PSSoDgYUAAoGBAO/C7ZQWHB6bT1W9NAwTEK+VsMXHe7cphNPA
YHhewy6kpNvfiuANtKF02FrmOtZ+N+sAC7GBrlviL5OBMdD/5j7/NeuMZsIMTjNZ4jLn5EzU/nux
bQPEhxxIngXj/Wh4It43MVDYBo0Z+pqjPL2vS/QBh4z8rz4ezOYGGxV2B3mw
私钥:
MIIBTAIBADCCASwGByqGSM44BAEwggEfAoGBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2
USZpRV1AIlH7WT2NWPq/xfW6MPbLm1Vs14E7gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4
O1fnxqimFQ8E+4P208UewwI1VBNaFpEy9nXzrith1yrv8iIDGZ3RSAHHAhUAl2BQjxUjC8yykrmC
```

更多资源请访问稀酷客(www.ckook.com)

```
ouuEC/BYHPUCgYEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCB
gLRJFnEj6EwoFhO3zwkyjMim4TwWeotUfI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhR
kImog9/hWuWfBpKLZl6Ae1UlZAFMO/7PSSoEFwIVAIB5fpdgw0p7LP/M8X91VOidqLej
```

注意控制台输出的签名及验证结果，如下所示：

```
签名：
302d021409e9d5a41a479a2ec08151fd21b6965a8f7d803e0215008b18632b91b6cd1d07a9f8db40
bc98a65144c89a
状态：
true
```

DSA算法签名长度与密钥长度无关，但可能与待签名数据存在某种关联。

9.5 椭圆曲线数字签名算法——ECDSA

对微软（Microsoft）产品有所了解的读者朋友可能对于这个算法的名称不会太陌生，它正是微软操作系统及办公软件的序列号验证算法。序列号是什么？正是微软为其软件经过签名得到的签名值！

9.5.1 简述

ECDSA（Elliptic Curve Digital Signature Algorithm，椭圆曲线数字签名算法）于1999年作为ANSI标准，并于2000年成为IEEE和NIST的标准。

ECDSA算法相对于传统签名算法具有速度快、强度高、签名短等优点，其用途也越来越广泛。微软操作系统的25位的产品密钥中就使用了椭圆曲线签名算法，产品密钥就是签名的十六进制串表示形式。

9.5.2 实现

Java 6对数字签名算法的支持很有限，除了RSA算法就只有DSA算法了。作者曾寻觅椭圆曲线加密算法许久，但至今未果，在研究Bouncy Castle时意外发现了椭圆曲线数字签名算法ECDSA。

Bouncy Castle支持多种数字签名算法，单纯ECDSA算法系列包括NONEwithECDSA、RIPEMD160withECDSA、SHA1withECDSA、SHA224withECDSA、SHA256withECDSA、SHA384withECDSA和SHA512withECDSA共7种算法。除此之外，Bouncy Castle还支持其他RSA数字签名算法，如SHA224withRSA、SHA256withRSA、SHA384withRSA和SHA512withRSA等。

ECDSA数字签名算法的密钥算法同为“ECDSA”，实现方式与RSA和DSA算法产生密钥的方式相类似。

有关ECDSA算法的Bouncy Castle实现如表9-3所示。

更多资源请访问稀酷客(www.ckook.com)

表9-3 ECDSA算法

算法	密钥长度	密钥长度默认值	签名长度	备注
NONEwithECDSA	—	—	128	Bouncy Castle实现
RIPEMD160withECDSA			160	
SHA1withECDSA			160	
SHA224withECDSA			224	
SHA256withECDSA			256	
SHA384withECDSA			384	
SHA512withECDSA			512	

Bouncy Castle未给出ECDSA算法的密钥长度及默认值信息，ECDSA算法密钥生成仅能通过算法材料的方式产生。这也是ECDSA数字签名算法与RSA和DSA数字签名算法在实现方面的主要差异。ECDSA算法密钥构建实现如代码清单9-7所示。

代码清单9-7 ECDSA数字签名算法生成密钥

```
import java.security.Security;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
// 省略
// 加入BouncyCastleProvider支持
Security.addProvider(new BouncyCastleProvider());
BigInteger p = new BigInteger
("883423532389192164791648750360308885314476597252960362792450860609699839");
ECFieldFp ecFieldFp = new ECFieldFp(p);
BigInteger a = new BigInteger
("7fffffffffffffffffffffff7fffffffffff8000000000007ffffffffc", 16);
BigInteger b = new BigInteger
("6b016c3bdcf18941d0d654921475ca71a9db2fb27d1d37796185c2942c0a", 16);
EllipticCurve ellipticCurve = new EllipticCurve(ecFieldFp, a, b);
BigInteger x = new BigInteger
("1102820037495488564763485335411862045779050615048812422401495115944209 11");
BigInteger y = new BigInteger
("869078407435509378747351873793058868500210384946040694651368759217025454");
ECPoint g = new ECPoint(x, y);
BigInteger n = new BigInteger
("883423532389192164791648750360308884807550341691627752275345424702807307");
ECParameterSpec ecParameterSpec = new ECParameterSpec(ellipticCurve, g, n, 1);
// 实例化密钥对生成器
KeyPairGenerator kpg = KeyPairGenerator.getInstance(KEY_ALGORITHM);
// 初始化密钥对生成器
kpg.initialize(ecParameterSpec, new SecureRandom());
```

作者按照Bouncy Castle提供的资料构建了上述密钥生成代码，该代码主要是为了产生算法参数ECParameterSpec实例ecParameterSpec对象，即构建ECC算法材料。

很遗憾，Bouncy Castle提供的ECDSA算法未能提供自动化密钥生成实现，这可能跟ECC算

更多资源请访问稀酷客(www.ckook.com)

法自身的实现难度较高有关。

本书旨在介绍Bouncy Castle对于数字签名算法的支持，有关ECC算法的Java API请读者朋友阅读相关文档。完整代码实现如代码清单9-8所示。

代码清单9-8 ECDSA算法实现

```
import java.math.BigInteger;
import java.security.Key;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.SecureRandom;
import java.security.Security;
import java.security.Signature;
import java.security.interfaces.ECPrivateKey;
import java.security.interfaces.ECPublicKey;
import java.security.spec.ECFieldFp;
import java.security.spec.ECParameterSpec;
import java.security.spec.ECPoint;
import java.security.spec.EllipticCurve;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.util.HashMap;
import java.util.Map;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
/**
 * ECDSA安全编码组件
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public abstract class ECDSACoder {
    /**
     * 数字签名
     * 密钥算法
     */
    private static final String KEY_ALGORITHM = "ECDSA";
    /**
     * 数字签名
     * 签名/验证算法
     * Bouncy Castle支持以下7种算法
     * NONEwithECDSA
     * RIPEMD160withECDSA
     * SHA1withECDSA
     * SHA224withECDSA
     * SHA256withECDSA
```

更多资源请访问稀酷客(www.ckook.com)

```
     * SHA384withECDSA
     * SHA512withECDSA
     */
    private static final String SIGNATURE_ALGORITHM = "SHA512withECDSA";
    // 公钥
    private static final String PUBLIC_KEY = "ECDSAPublicKey";
    // 私钥
    private static final String PRIVATE_KEY = "ECDSAPrivateKey";
    /**
     * 初始化密钥
     * @return Map 密钥Map
     * @throws Exception
     */
    public static Map<String, Object> initKey() throws Exception {
        // 加入BouncyCastleProvider支持
        Security.addProvider(new BouncyCastleProvider());
        BigInteger p = new BigInteger
        ("883423532389192164791648750360308885314476597252960362792450860609699839");
        ECFieldFp ecFieldFp = new ECFieldFp(p);
        BigInteger a = new BigInteger
        ("7fffffffffffffffffffffff7fffffffffff8000000000007ffffffffffc", 16);
        BigInteger b = new BigInteger
        (6b016c3bdcf18941d0d654921475ca71a9db2fb27d1d37796185c2942c0a", 16);
        EllipticCurve ellipticCurve = new EllipticCurve(ecFieldFp, a, b);
        BigInteger x = new BigInteger
        ("1102820037495488564763485335411862045779050615048812422401495115944
        20911");
        BigInteger y = new BigInteger
        ("8690784074355093787473518737930588685002103849460406946513687592170
        25454");
        ECPoint g = new ECPoint(x, y);
        BigInteger n = new BigInteger
        ("8834235323891921647916487503603088848075503416916277522753454247028
        07307");
        ECParameterSpec ecParameterSpec = new ECParameterSpec(ellipticCurve,
        g, n, 1);
        // 实例化密钥对生成器
        KeyPairGenerator kpg = KeyPairGenerator.getInstance(KEY_ALGORITHM);
        // 初始化密钥对生成器
        kpg.initialize(ecParameterSpec, new SecureRandom());
        // 生成密钥对
        KeyPair keypair = kpg.generateKeyPair();
        ECPublicKey publicKey = (ECPublicKey) keypair.getPublic();
        ECPrivateKey privateKey = (ECPrivateKey) keypair.getPrivate();
        // 封装密钥
        Map<String, Object> map = new HashMap<String, Object>(2);
        map.put(PUBLIC_KEY, publicKey);
```

更多资源请访问稀酷客(www.ckook.com)

```
        map.put(PRIVATE_KEY, privateKey);
        return map;
    }
    /**
     * 取得私钥
     * @param keyMap 密钥Map
     * @return byte[] 私钥
     * @throws Exception
     */
    public static byte[] getPrivateKey(Map<String, Object> keyMap) throws Exception {
        Key key = (Key) keyMap.get(PRIVATE_KEY);
        return key.getEncoded();
    }
    /**
     * 取得公钥
     * @param keyMap 密钥Map
     * @return byte[] 公钥
     * @throws Exception
     */
    public static byte[] getPublicKey(Map<String, Object> keyMap) throws Exception {
        Key key = (Key) keyMap.get(PUBLIC_KEY);
        return key.getEncoded();
    }
    /**
     * 签名
     * @param data 待签名数据
     * @param privateKey 私钥
     * @return byte[] 数字签名
     * @throws Exception
     */
    public static byte[] sign(byte[] data, byte[] privateKey) throws Exception {
        // 加入BouncyCastleProvider支持
        Security.addProvider(new BouncyCastleProvider());
        // 转换私钥材料
        PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(privateKey);
        // 实例化密钥工厂
        KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
        // 取私钥匙对象
        PrivateKey priKey = keyFactory.generatePrivate(pkcs8KeySpec);
        // 实例化Signature
        Signature signature = Signature.getInstance(SIGNATURE_ALGORITHM);
        // 初始化Signature
        signature.initSign(priKey);
        // 更新
        signature.update(data);
        // 签名
        return signature.sign();
    }
```

更多资源请访问稀酷客(www.ckook.com)

```
/**
 * 校验
 * @param data 待校验数据
 * @param publicKey 公钥
 * @param sign 数字签名
 * @return boolean 校验成功返回true 失败返回false
 * @throws Exception
 */
public static boolean verify(byte[] data, byte[] publicKey, byte[] sign)
throws Exception {
        // 加入BouncyCastleProvider支持
        Security.addProvider(new BouncyCastleProvider());
        // 转换公钥材料
        X509EncodedKeySpec keySpec = new X509EncodedKeySpec(publicKey);
        // 实例化密钥工厂
        KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
        // 生成公钥
        PublicKey pubKey = keyFactory.generatePublic(keySpec);
        // 实例化Signature
        Signature signature = Signature.getInstance(SIGNATURE_ALGORITHM);
        // 初始化Signature
        signature.initVerify(pubKey);
        // 更新
        signature.update(data);
        // 验证
        return signature.verify(sign);
    }
}
```

对于上述代码的校验与其他数字签名算法无异，如代码清单9-9所示。

代码清单9-9 ECDSA算法实现测试用例

```
import static org.junit.Assert.*;
import java.util.Map;
import org.apache.commons.codec.binary.Base64;
import org.apache.commons.codec.binary.Hex;
import org.junit.Before;
import org.junit.Test;
/**
 * ECDSA数字签名校验
 * @author 梁栋
 * @version 1.0
 */
public class ECDSACoderTest {
    // 公钥
    private byte[] publicKey;
    // 私钥
    private byte[] privateKey;
```

更多资源请访问稀酷客(www.ckook.com)

```
    /**
     * 初始化密钥
     * @throws Exception
     */
    @Before
    public void initKey() throws Exception {
            Map<String, Object> keyMap = ECDSACoder.initKey();
            publicKey = ECDSACoder.getPublicKey(keyMap);
            privateKey = ECDSACoder.getPrivateKey(keyMap);
            System.err.println("公钥: \n" + Base64.encodeBase64String(publicKey));
            System.err.println("私钥: \n" + Base64.encodeBase64String(privateKey));
    }
    /**
     * 校验
     * @throws Exception
     */
    @Test
    public void test() throws Exception {
            String inputStr = "ECDSA数字签名";
            byte[] data = inputStr.getBytes();
            // 产生签名
            byte[] sign = ECDSACoder.sign(data, privateKey);
            System.err.println("签名:\r" + Hex.encodeHexString(sign));
            // 验证签名
            boolean status = ECDSACoder.verify(data, publicKey, sign);
            System.err.println("状态:\r" + status);
            // 验证
            assertTrue(status);
        }
    }
```

ECDSA算法与DSA算法共同的特点在于公钥和密钥长度上的差异，ECDSA算法密钥同样是公钥长度略长于私钥，如下所示：

```
公钥:
MIIBITCB3gYHKoZIzj0CATCB0gIBATApBgcqhkjOPQEBAh5/////////////////9///////+AAAAA
AAB///////8wQAQef////////////////f///////gAAAAAAAf//////8BB5rAWw73PGJQdDWVJIU
dcpxqdsvsn0dN3lhhcKULAoEPQQP+pY83KiBbMwzuGQr7fkFw9NYVz0/J/u9Ozy5qq996+jk6Qpd
rm5AVMpTC6BGVLNoGM4iazn8y3sC8a4CHn/////////////////3///55emp9dkHH70VImiJCdCwIB
AQM+AARVICaPidHhLwSf3Vu8URtBaOleb6csh0cmWhBAqkp/hHGCIaN9iMbUyhxUrO3UyegLZvN6
wGgBYK7cIy0=
私钥:
MIIBCwIBADCB3gYHKoZIzj0CATCB0gIBATApBgcqhkjOPQEBAh5/////////////////9///////+A
AAAAAAB///////8wQAQef////////////////f///////gAAAAAAAf//////8BB5rAWw73PGJQdDW
VJIUdcpxqdsvsn0dN3lhhcKULAoEPQQP+pY83KiBbMwzuGQr7fkFw9NYVz0/J/u9Ozy5qq996+jk
6Qpdrm5AVMpTC6BGVLNoGM4iazn8y3sC8a4CHn/////////////////3///55emp9dkHH70VImiJCd
CwIBAQQlMCMCAQEEHi4tPTGcWH8OxoPquDEuYTmmobCqO/GC696avIFEJw==
```

更多资源请访问稀酷客(www.ckook.com)

签名验证结果如下所示：

```
签名：
3040021e66913eccf0cb3ccb16b224ee6c53069ccea90a3da89d113c7d585bcedd35021e2707af24
42e2d14ad68b364a0a068e39896daed5daed09df676b8e521e0c
状态：
true
```

作者根据Bouncy Castle提供的资料构建了ECDSA算法实现，其密钥的产生主要依赖于算法参数。ECDSA算法参数构建复杂，需要对算法相关的数学公式有一定的了解。上述签名长度是否与密钥长度或待签名信息有关，我们无从得知。

9.6 实例：带有数字签名的加密网络应用

在第8章构建DataServer应用时，我们给读者朋友留下一个问题：如何使用数字签名验证数据。这里我们不再使用消息摘要算法对数据进行摘要/验证，取而代之使用RSA签名算法对其进行签名/验证，但其实质都是通过消息摘要算法对数据进行摘要/验证。

我们继续对第8章构建的DataServer应用进行演进，使用RSA签名算法对服务器下发的数据进行签名，并在测试用例中进行验证。

首先，我们需要将第8章的RSACoder类和本章的RSACoder类进行合并。简单来讲就是将本章RSACoder类的sign()和verify()方法合并到第8章的RSACoder类。本书为了详述各个算法的具体实现将这两种RSA算法进行了分离，在实际应用中这两种算法都是一起配合使用的。

接下来，我们将使用开源组件Commons Codec的十六进制转换工具类Hex对签名进行封装/解包。相关实现如代码清单9-10所示。

代码清单9-10 RSACoder——签名封装/解包

```
/**
 * 私钥签名
 * @param data 待签名数据
 * @param privateKey 私钥
 * @return String 十六进制签名字符串
 * @throws Exception
 */
public static String sign(byte[] data, String privateKey) throws Exception {
    byte[] sign = sign(data, getKey(privateKey));
    return Hex.encodeHexString(sign);
}
/**
 * 公钥校验
 * @param data 待验证数据
 * @param publicKey 公钥
 * @param sign 签名
 * @return boolean 成功返回true，失败返回false
```

更多资源请访问稀酷客(www.ckook.com)

```
     * @throws Exception
     */
    public static boolean verify(byte[] data, String publicKey, String sign) throws Exception {
        return verify(data, getKey(publicKey), Hex.decodeHex(sign.toCharArray()));
    }
```

完整实现如代码清单9-11所示。

代码清单9-11 RSACoder

```
import java.security.Key;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.Signature;
import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.util.HashMap;
import java.util.Map;
import javax.crypto.Cipher;
import org.apache.commons.codec.binary.Hex;
/**
 * RSA安全编码组件
 * @author 梁栋
 * @version 1.0
 */
public abstract class RSACoder {
    // 非对称加密密钥算法
    public static final String KEY_ALGORITHM = "RSA";
    // 数字签名 签名/验证算法
    public static final String SIGNATURE_ALGORITHM = "SHA1withRSA";
    // 公钥
    private static final String PUBLIC_KEY = "RSAPublicKey";
    // 私钥
    private static final String PRIVATE_KEY = "RSAPrivateKey";
    /**
     * RSA密钥长度 默认1024位，密钥长度必须是64的倍数，范围在512～65536位之间。
     */
    private static final int KEY_SIZE = 512;
    /**
     * 私钥解密
     * @param data 待解密数据
     * @param key 私钥
     * @return byte[] 解密数据
     * @throws Exception
```

更多资源请访问稀酷客(www.ckook.com)

```
     */
    public static byte[] decryptByPrivateKey(byte[] data, byte[] key) throws Exception {
            // 取得私钥
            PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(key);
            KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
            // 生成私钥
            PrivateKey privateKey = keyFactory.generatePrivate(pkcs8KeySpec);
            // 对数据解密
            Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
            cipher.init(Cipher.DECRYPT_MODE, privateKey);
            return cipher.doFinal(data);
    }
    /**
     * 公钥解密
     * @param data 待解密数据
     * @param key 公钥
     * @return byte[] 解密数据
     * @throws Exception
     */
    public static byte[] decryptByPublicKey(byte[] data, byte[] key) throws Exception {
            // 取得公钥
            X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(key);
            KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
            // 生成公钥
            PublicKey publicKey = keyFactory.generatePublic(x509KeySpec);
            // 对数据解密
            Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
            cipher.init(Cipher.DECRYPT_MODE, publicKey);
            return cipher.doFinal(data);
    }
    /**
     * 公钥加密
     * @param data 待加密数据
     * @param key 公钥
     * @return byte[] 加密数据
     * @throws Exception
     */
    public static byte[] encryptByPublicKey(byte[] data, byte[] key) throws Exception {
            // 取得公钥
            X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(key);
            KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
            PublicKey publicKey = keyFactory.generatePublic(x509KeySpec);
            // 对数据加密
            Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
            cipher.init(Cipher.ENCRYPT_MODE, publicKey);
            return cipher.doFinal(data);
    }
    /**
```

更多资源请访问稀酷客(www.ckook.com)

```
     * 私钥加密
     * @param data 待加密数据
     * @param key 私钥
     * @return byte[] 加密数据
     * @throws Exception
     */
    public static byte[] encryptByPrivateKey(byte[] data, byte[] key) throws Exception {
            // 取得私钥
            PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(key);
            KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
            // 生成私钥
            PrivateKey privateKey = keyFactory.generatePrivate(pkcs8KeySpec);
            // 对数据加密
            Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
            cipher.init(Cipher.ENCRYPT_MODE, privateKey);
            return cipher.doFinal(data);
    }
    /**
     * 取得私钥
     * @param keyMap 密钥Map
     * @return byte[] 私钥
     * @throws Exception
     */
    public static byte[] getPrivateKey(Map<String, Object> keyMap) throws Exception {
            Key key = (Key) keyMap.get(PRIVATE_KEY);
            return key.getEncoded();
    }
    /**
     * 取得公钥
     * @param keyMap 密钥Map
     * @return byte[] 公钥
     * @throws Exception
     */
    public static byte[] getPublicKey(Map<String, Object> keyMap)
                    throws Exception {
            Key key = (Key) keyMap.get(PUBLIC_KEY);
            return key.getEncoded();
    }
    /**
     * 初始化密钥
     * @return Map 密钥Map
     * @throws Exception
     */
    public static Map<String, Object> initKey() throws Exception {
            // 实例化密钥对生成器
            KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance (KEY_ALGORITHM);
            // 初始化密钥对生成器
            keyPairGen.initialize(KEY_SIZE);
```

更多资源请访问稀酷客(www.ckook.com)

```
        // 生成密钥对
        KeyPair keyPair = keyPairGen.generateKeyPair();
        // 公钥
        RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();
        // 私钥
        RSAPrivateKey privateKey = (RSAPrivateKey) keyPair.getPrivate();
        // 封装密钥
        Map<String, Object> keyMap = new HashMap<String, Object>(2);
        keyMap.put(PUBLIC_KEY, publicKey);
        keyMap.put(PRIVATE_KEY, privateKey);
        return keyMap;
    }
    /**
     * 签名
     * @param data 待签名数据
     * @param privateKey 私钥
     * @return byte[] 数字签名
     * @throws Exception
     */
    public static byte[] sign(byte[] data, byte[] privateKey) throws Exception {
        // 转换私钥材料
        PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(privateKey);
        // 实例化密钥工厂
        KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
        // 取私钥对象
        PrivateKey priKey = keyFactory.generatePrivate(pkcs8KeySpec);
        // 实例化Signature
        Signature signature = Signature.getInstance(SIGNATURE_ALGORITHM);
        // 初始化Signature
        signature.initSign(priKey);
        // 更新
        signature.update(data);
        // 签名
        return signature.sign();
    }
    /**
     * 公钥校验
     * @param data 待校验数据
     * @param publicKey 公钥
     * @param sign 数字签名
     * @return boolean 校验成功返回true 失败返回false
     * @throws Exception
     */
    public static boolean verify(byte[] data, byte[] publicKey, byte[] sign)
    throws Exception {
        // 转换公钥材料
        X509EncodedKeySpec keySpec = new X509EncodedKeySpec(publicKey);
        // 实例化密钥工厂
```

更多资源请访问稀酷客(www.ckook.com)

```
        KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
        // 生成公钥
        PublicKey pubKey = keyFactory.generatePublic(keySpec);
        // 实例化Signature
        Signature signature = Signature.getInstance(SIGNATURE_ALGORITHM);
        // 初始化Signature
        signature.initVerify(pubKey);
        // 更新
        signature.update(data);
        // 验证
        return signature.verify(sign);
}
/**
 * 私钥签名
 * @param data 待签名数据
 * @param privateKey 私钥
 * @return String 十六进制签名字符串
 * @throws Exception
 */
public static String sign(byte[] data, String privateKey) throws Exception {
        byte[] sign = sign(data, getKey(privateKey));
        return Hex.encodeHexString(sign);
}
/**
 * 公钥校验
 * @param data 待验证数据
 * @param publicKey 公钥
 * @param sign 签名
 * @return boolean 成功返回true，失败返回false
 * @throws Exception
 */
public static boolean verify(byte[] data, String publicKey, String sign)
throws Exception {
        return verify(data, getKey(publicKey), Hex.decodeHex(sign.toCharArray()));
}
/**
 * 私钥加密
 * @param data 待加密数据
 * @param key 私钥
 * @return byte[] 加密数据
 * @throws Exception
 */
public static byte[] encryptByPrivateKey(byte[] data, String key) throws Exception {
        return encryptByPrivateKey(data, getKey(key));
}
/**
 * 公钥加密
 * @param data 待加密数据
```

更多资源请访问稀酷客(www.ckook.com)

```
     * @param key 公钥
     * @return byte[] 加密数据
     * @throws Exception
     */
    public static byte[] encryptByPublicKey(byte[] data, String key) throws Exception {
            return encryptByPublicKey(data, getKey(key));
    }
    /**
     * 私钥解密
     * @param data 待解密数据
     * @param key 私钥
     * @return byte[] 解密数据
     * @throws Exception
     */
    public static byte[] decryptByPrivateKey(byte[] data, String key) throws Exception {
            return decryptByPrivateKey(data, getKey(key));
    }
    /**
     * 公钥解密
     * @param data 待解密数据
     * @param key 私钥
     * @return byte[] 解密数据
     * @throws Exception
     */
    public static byte[] decryptByPublicKey(byte[] data, String key) throws Exception {
            return decryptByPublicKey(data, getKey(key));
    }
    /**
     * 初始化密钥
     * @param keyMap 密钥Map
     * @return String 十六进制编码密钥
     * @throws Exception
     */
    public static String getPrivateKeyString(Map<String, Object> keyMap) throws Exception {
            return Hex.encodeHexString(getPrivateKey(keyMap));
    }
    /**
     * 初始化密钥
     * @param keyMap 密钥Map
     * @return String 十六进制编码密钥
     * @throws Exception
     */
    public static String getPublicKeyString(Map<String, Object> keyMap) throws Exception {
            return Hex.encodeHexString(getPublicKey(keyMap));
    }
    /**
     * 获取密钥
     * @param key 密钥
```

更多资源请访问稀酷客(www.ckook.com)

```
     * @return byte[] 密钥
     * @throws Exception
     */
    public static byte[] getKey(String key) throws Exception {
            return Hex.decodeHex(key.toCharArray());
    }
}
```

接下来，我们将对DataServlet类进行改造，使其下发数据时附带该数据签名串。

我们可以使用RSACoder类的sign()方法对数据进行签名，并在控制台输出该签名。相关实现如代码清单9-12所示。

代码清单9-12 DataServlet——构建签名

```
// 使用RSA签名算法对数据签名
String sign = RSACoder.sign(output, key);
System.err.println("sign : \r\n" + sign);
```

通过本章阐述，我们知道RSA签名串长度与初始化密钥长度一致。因此，我们可以将该签名串与原数据进行串连。此处，我们需要将签名串作为数据包顶部数据。最终使用AES算法将该数据包加密并回复给调用方。相关实现如代码清单9-13所示。

代码清单9-13 DataServlet——数据包处理

```
/*
 * 重新组织待输出的数据，将该数据的签名信息封装在数据包顶部
 * RSA数字签名得到的签名值长度固定，与初始化密钥长度相同。
 * 如初始化密钥长度为512位，且使用RSAwithSHA1算法，
 * 则得到的签名串长度为128位十六进制串，即512位二进制数。
 */
ByteArrayOutputStream baos = new ByteArrayOutputStream();
// 此处将写入128位十六进制字符
baos.write(sign.getBytes());
baos.write(output);
output = baos.toByteArray();
baos.flush();
baos.close();
// 使用AES算法对数据加密并回复
HttpUtils.responseWrite(response, AESCoder.encrypt(output, k));
```

完整实现如代码清单9-14所示。

代码清单9-14 DataServlet

```
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

更多资源请访问稀酷客(www.ckook.com)

```
/**
 * 数据服务DataServlet
 * @author 梁栋
 * @since 1.0
 */
public class DataServlet extends HttpServlet {
    private static final long serialVersionUID = -6219906900195793155L;
    // 密钥
    private static String key;
    // Servlet初始化参数——私钥
    private static final String KEY_PARAM = "key";
    // 初始化
    @Override
    public void init() throws ServletException {
        super.init();
        // 初始化密钥
        key = getInitParameter(KEY_PARAM);
    }
    // 处理POST请求
    protected void doPost(HttpServletRequest request,
            HttpServletResponse response) throws ServletException,
            IOException {
        try {
            byte[] input = HttpUtils.requestRead(request);
            // 对秘密密钥解密
            String k = new String(RSACoder.decryptByPrivateKey(input, key));
            // 输出秘密密钥
            System.err.println(k);
            // 构造数据包
            StringBuilder sb = new StringBuilder();
            sb.append("<?xml version=\"1.0\" encoding=\"UTF-8\"?>\r\n");
            sb.append("<dataGroup>\r\n");
            sb.append("\t<dataItem>\r\n");
            sb.append("\t\t<id>");
            sb.append("10201");
            sb.append("</id>\r\n");
            sb.append("\t\t<price>");
            sb.append("35.0");
            sb.append("</price>\r\n");
            sb.append("\t\t<time>");
            sb.append("2009-10-30");
            sb.append("</time>\r\n");
            sb.append("\t</dataItem>\r\n");
            sb.append("\t<dataItem>\r\n");
            sb.append("\t\t<id>");
            sb.append("10301");
            sb.append("</id>\r\n");
            sb.append("\t\t<price>");
```

更多资源请访问稀酷客(www.ckook.com)

```
                sb.append("55.0");
                sb.append("</price>\r\n");
                sb.append("\t\t<time>");
                sb.append("2009-10-31");
                sb.append("</time>\r\n");
                sb.append("\t</dataItem>\r\n");
                sb.append("</dataGroup>\r\n");
                byte[] output = sb.toString().getBytes();
                // 使用RSA签名算法对数据签名
                String sign = RSACoder.sign(output, key);
                System.err.println("sign : \r\n" + sign);
                /*
                 * 重新组织待输出的数据，将该数据的签名信息封装在数据包顶部
                 * RSA数字签名得到的签名值长度固定，与初始化密钥长度相同。
                 * 如初始化密钥长度为512位，且使用RSAwithSHA1算法，
                 * 则得到的签名串长度为128位十六进制串，即512位二进制数。
                */
                ByteArrayOutputStream baos = new ByteArrayOutputStream();
                // 此处将写入128位十六进制字符
                baos.write(sign.getBytes());
                baos.write(output);
                output = baos.toByteArray();
                baos.flush();
                baos.close();
                // 使用AES算法对数据加密并回复
                HttpUtils.responseWrite(response, AESCoder.encrypt(output, k));
            } catch (Exception e) {
                    throw new ServletException(e);
            }
        }
    }
```

此时，我们同样需要修改测试用例DataServletTest类相关代码，使其对服务器回应的数据进行验证操作。

这里，我们需要使用AESCoder的decrypt()方法对数据解密，并根据我们一致的签名长度对解析后的数据进行分离，得到签名及数据，并将其打印的控制台。相关实现如代码清单9-15所示。

代码清单9-15　DataServletTest——签名、数据分离

```
// 使用AES算法对数据解密
String data = new String(AESCoder.decrypt(input, secretKey));
// 分离签名、数据
String sign = data.substring(0, 128);
data = data.substring(128);
System.err.println("sign: \r\n" + sign);
System.err.println("data: \r\n" + data);
```

接着，我们可以使用RSACoder类的verify()方法对签名进行验证。相关实现如代码清单9-16所示。

更多资源请访问稀酷客(www.ckook.com)

代码清单9-16 DataServletTest——签名验证

```
// 校验
assertTrue(RSACoder.verify(data.getBytes(), publicKey, sign));
```

DataServletTest完整实现如代码清单9-17所示。

代码清单9-17 DataServletTest

```
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * DataServlet测试用例
 * @author 梁栋
 * @since 1.0
 */
public class DataServletTest {
    // 公钥
    private static final String publicKey =
    "305c300d06092a864886f70d0101010500034b0030480241009fec6cff0209ef1a332a35cca
    fc2aae59c4d5275ef9186d73593186482ec637f6df042c2aa41115c8a1625a2e9e9f7844c008
    389e5a6379a268d877fd8edc2690203010001";
    // 请求地址
    private static final String url =
    "http://localhost:8080/dataserver/DataServlet";
        @Test
        public final void test() throws Exception {
                // 构建秘密密钥
                String secretKey = AESCoder.initKeyString();
                // 使用RSA算法加密并发送秘密密钥
                byte[] input = HttpUtils.postRequest(url, RSACoder.encryptByPublicKey
                (secretKey.getBytes(), publicKey));
                // 使用AES算法对数据解密
                String data = new String(AESCoder.decrypt(input, secretKey));
                // 分离签名、数据
                String sign = data.substring(0, 128);
                data = data.substring(128);
                System.err.println("sign: \r\n" + sign);
                System.err.println("data: \r\n" + data);
                // 校验
                assertTrue(RSACoder.verify(data.getBytes(), publicKey, sign));
        }
    }
```

启动DataServer服务，并执行DataServletTest类的测试方法，我们将在控制台中得到签名和数据内容，如下所示：

```
sign:
7f9d7535662b099f202e71b1614e7eb1ce0341e16eead56bd26e8aa0a1d4e71992f092f02f058314
b052cb8f495823687b2effb86bf1c19f071985d5c77e8a43
```

更多资源请访问稀酷客(www.ckook.com)

```
data:
<?xml version="1.0" encoding="UTF-8"?>
<dataGroup>
    <dataItem>
            <id>10201</id>
            <price>35.0</price>
            <time>2009-10-30</time>
    </dataItem>
    <dataItem>
            <id>10301</id>
            <price>55.0</price>
            <time>2009-10-31</time>
    </dataItem>
</dataGroup>
```

如果验证失败，我们则认为数据在传输过程中被篡改，本次交互失败。

在实际应用中，很少会直接使用数字签名对数据进行签名/验证。并且，较少直接使用非对称加密算法对数据进行加密/解密。这些操作通常叫做底层操作，可配合数字证书、SSL/TLS协议构建单向认证或双向认证使用。

本书第6～9章详述了消息摘要算法、对称加密算法、非对称加密算法和数字签名算法共4大类加密算法，这些算法最终被有机地组合，以构建单向/双向认证服务，请读者朋友关注后续章节相关内容。

9.7 小结

数字签名算法是公钥基础设施（Public Key Infrastructure，PKI）以及许多网络安全机制（SSL/TLS、VPN等）的基础。数字签名算法包含签名和验证两项操作，遵循“私钥签名，公钥验证”的签名/验证方式，签名时需要使用私钥和待签名数据，其核心算法主要是消息摘要算法。因此，我们可以把数字签名算法近似看成是一种附加了公钥和私钥的消息摘要算法。

数字签名算法主要包括RSA、DSA和ECDSA共3种算法。其中，RSA算法源于整数因子分解问题，DSA和ECDSA算法源于离散对数问题。

RSA算法是数字签名算法中的经典，主要可以分为MD系列和SHA系列两大类。

RSA算法是目前应用最为广泛的非对称加密算法和数字签名算法，在电子商务和产品验证方面均有使用。

DSA算法是继RSA算法后出现的基于DSS的数字签名算法，旨在形成数字签名标准。并且，DSA算法本身不包含任何消息摘要算法。DSA算法主要为后续数字签名算法的形成奠定基础。

Java 6提供了DSA算法实现，在实现层面，我们可以认为DSA算法实现就是RSA数字签名算法实现的简装版。

ECDSA算法相对传统签名算法具有速度快、强度高、签名短等优点，其用途也越来越广泛。微软操作系统的25位的产品密钥中就使用了椭圆曲线签名算法，产品密钥就是签名的十六进制串表示形式。

更多资源请访问稀酷客(www.ckook.com)

更多资源请访问稀酷客(www.ckook.com)

Part3 第三部分

综合应用篇

第10章 终极武器——数字证书
第11章 终极装备——安全协议
第12章 量体裁衣——为应用选择合适的装备

更多资源请访问稀酷客(www.ckook.com)

第10章

终极武器——数字证书

本书阐述了大量的密码学算法，用于实现OSI参考模型中的安全服务：消息摘要算法用于验证数据完整性服务，对称加密算法和非对称加密算法用于保证数据保密性服务，数字签名算法用于抗否认性服务。

安全，向来不是一个简单的问题！如果仅仅使用一种单纯的防御手段，很难达到我们所期望的安全强度！安全的网络平台，需要多重密码学算法的有机结合！

数字证书集合了多种密码学算法：自身带有公钥信息，可完成相应的加密/解密操作；同时，还带有数字签名，可鉴别消息来源；且自身带有消息摘要信息，可验证证书的完整性；由于证书本身含有用户身份信息，因而具有认证性。

10.1 数字证书详解

数字证书具备常规加密/解密必要的信息，包含签名算法，可用于网络数据加密/解密交互，标识网络用户（计算机）身份。数字证书为发布公钥提供了一种简便的途径，其数字证书则成为加密算法以及公钥的载体。依靠数字证书，我们可以构建一个简单的加密网络应用平台。

数字证书（Digital Certificate）也称为电子证书，类似于我们生活中的身份证，用于标识网络中的用户（计算机）身份。在现实生活中，我们的身份证需由公安机关的签发，而网络用户（计算机）的身份凭证则需由数字证书颁发认证机构（Certificate Authority，CA）签发，只有经过CA签发的证书在网络中才具备可认证性。

VeriSign（http://www.verisign.com/）、GeoTrust（http://www.geotrust.com/）和Thawte（http://www.thawte.com/）是国际权威数字证书颁发认证机构的“三巨头”。其中，应用最为广泛的是VeriSign签发的电子商务用数字证书。

通常，这种由国际权威数字证书颁发认证机构颁发的数字证书需要向用户收取昂贵的申请和维护费用。但并不是所有的国际权威数字证书颁发认证机构都收费，CAcert（http://www.cacert.org/）就是一个免费的数字证书颁发国际组织。随着用户群的增大和颁发手段的可信性，这种免费的数字证书可信度也越来越高。

我国在每个省、直辖市设置有国家权威的数字证书颁发机构，比如北京市数字证书认证中

更多资源请访问稀酷客(www.ckook.com)

心、安徽省数字证书认证中心、广东省电子商务认证中心等。

证书的签发过程实际上是对申请数字证书的公钥做数字签名，证书的验证过程实际上是对数字证书的公钥做验证签名，其中还包含证书有效期验证。

数字证书如图10-1所示，这是由CA（www.zlex.org）颁发给自己的数字证书，也称为“根证书”。

通过使用由CA颁发的数字证书，我们可以对网络上传输的数据进行加密/解密和签名/验证操作，确保数据的机密性、完整性和抗否认性。同时，数字证书中包含的用户信息可以明确地标识交易实体身份，具有认证性，保证交易实体身份的真实性，从而保障网络应用的安全性。

实际上，数字证书是采用了公钥基础设施（Public Key Infrastructure，PKI），使用了相应的加密算法确保网络应用的安全性：

- 非对称加密算法用于对数据进行加密/解密操作，确保数据的机密性。
- 数字签名算法对用于对数据进行签名/验证操作，确保数据的完整性和抗否认性。
- 消息摘要算法对用于对数字证书本身做摘要处理，确保数字证书完整性。

目前，数字证书中最为常用的非对称加密算法是RSA算法，与之配套使用的签名算法是SHA1withRSA算法，而最为常用的消息摘要算法是SHA1算法，相关信息如图10-2所示。

图10-2中的签名算法sha1RSA为SHA1withRSA缩写形式，指纹算法（SHA1）即为消息摘要算法。

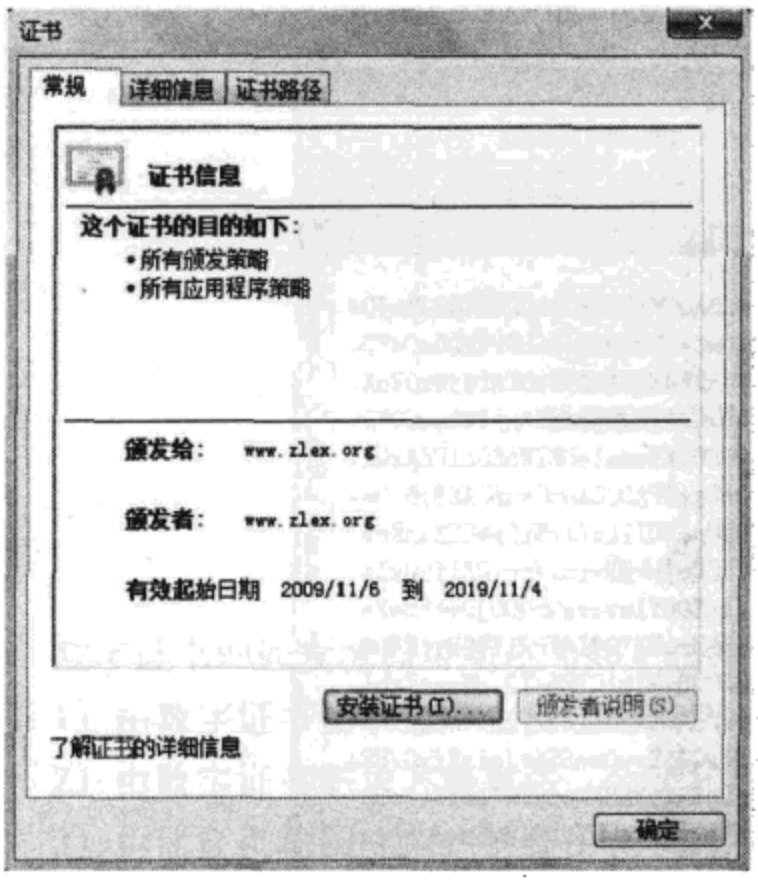


图10-1 数字证书

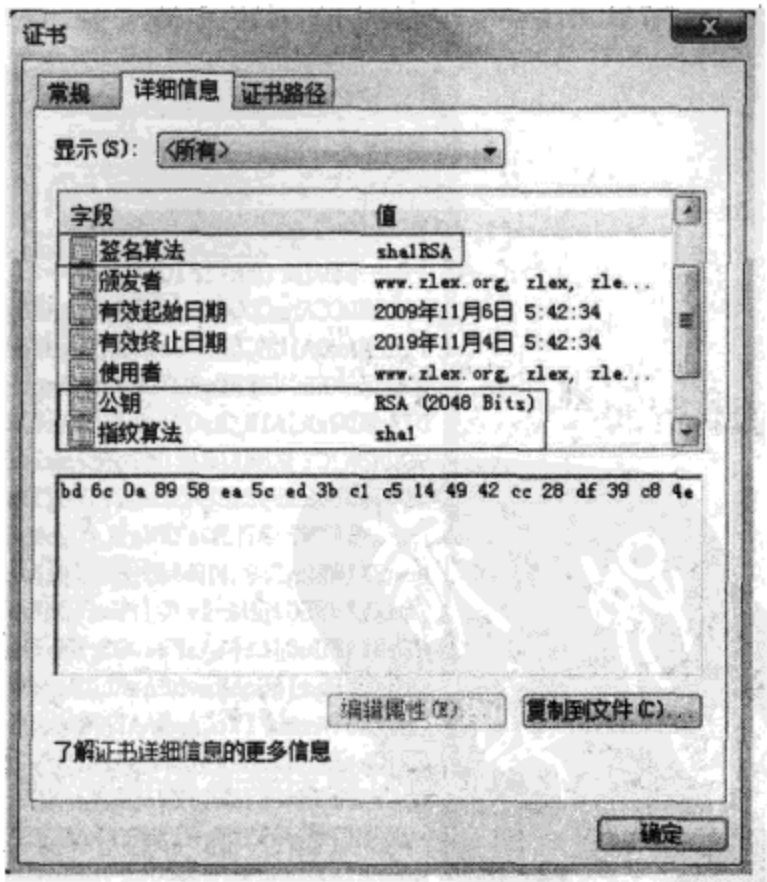


图10-2 数字证书的详细信息

除了使用RSA算法外，我们还可以使用DSA算法。只是使用DSA算法无法完成加密/解密实现，即这样的数字证书不包括数据加密/解密功能。

数字证书有多种文件编码格式，主要包含CER编码、DER编码等：

- CER（Canonical Encoding Rules，规范编码格式）是数字证书的一种编码格式，它是

更多资源请访问稀酷客(www.ckook.com)

BER（Basic Encoding Rules，基本编码格式）的一个变种，比BER规定得更严格。

- DER（Distinguished Encoding Rule，卓越编码格式）同样是BER的一个变种，与CER的不同之处在于：DER使用定长模式，而CER使用变长模式。

所有证书都符合公钥基础设施（PKI）制定的ITU-T X509国际标准（X.509标准），目前共包含3个版本。

- PKCS（Public-Key Cryptography Standards，公钥加密标准）由 RSA 实验室和其他安全系统开发商为促进公钥密码的发展而制定的一系列标准。

PKCS至今共发布过15个标准，常用标准主要包括PKCS#7、PKCS#10和PKCS#12。PKCS常用标准详见表10-1。

表10-1 PKCS常用标准

公钥加密标准	描述信息	文件名后缀
PKCS#7	密码消息语法标准	.p7b、.p7c、.spc
PKCS#10	证书请求语法标准	.p10、.csr
PKCS#12	个人信息交换语法标准	.p12、.pfx

以上标准主要用于证书的申请和更新等操作，例如，PKCS#10文件用于证书签发申请，PKCS#12文件可作为Java中的密钥库或信任库直接使用。

通常使用Base64编码格式作为数字证书文件存储格式，如图10-3所示。



图10-3 数字证书的Base64编码格式

更多资源请访问稀酷客(www.ckook.com)

在获得数字证书后，可以将其保存在电脑里，也可以保存在IC卡或USB Key中。

我们在使用银行借记卡或信用卡进行网上交易时，为增强网络数据传输安全性使用银行提供的“U盾”或其他设备。这种设备中实际上存储了银行提供的数字证书，并可通过设备上固化的程序对数字证书做定期升级。

10.2 模型分析

在实际应用中，很多数字证书都属于自签名证书，即证书申请者为自己的证书签名。这类证书通常应用于软件厂商内部发放的产品中，或约定使用该证书的数据交互双方。数字证书完全充当加密算法的载体，为必要数据做加密/解密和签名/验证等操作。

本文以使用数字证书进行加密消息传递为例，介绍证书签发和加密交互操作。

10.2.1 证书签发

数字证书需要经由认证机构签发，其流程如图10-4所示。

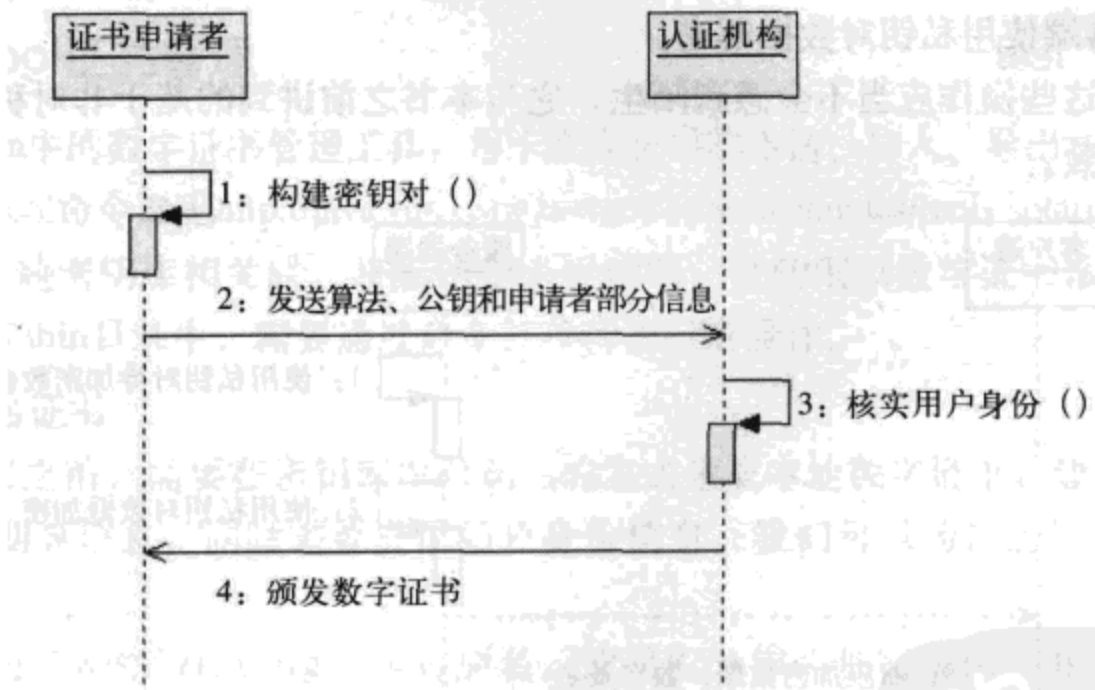


图10-4 数字证书颁发流程

数字证书的颁发流程可简述为如下过程：

1）由数字证书需求方产生自己的密钥对。

2）由数字证书需求方将算法、公钥和证书申请者身份信息传送给认证机构。

3）由认证机构核实用户的身份，执行相应必要的步骤，确保请求确实由用户发送而来。

4）由认证机构将数字证书颁发给用户。

这里的认证机构如果是证书申请者本身，将获得自签名证书。

10.2.2 加密交互

当客户端获得服务器下发的数字证书后，即可使用数字证书进行加密交互，分别如图10-5

更多资源请访问稀酷客(www.ckook.com)

和图10-6所示。

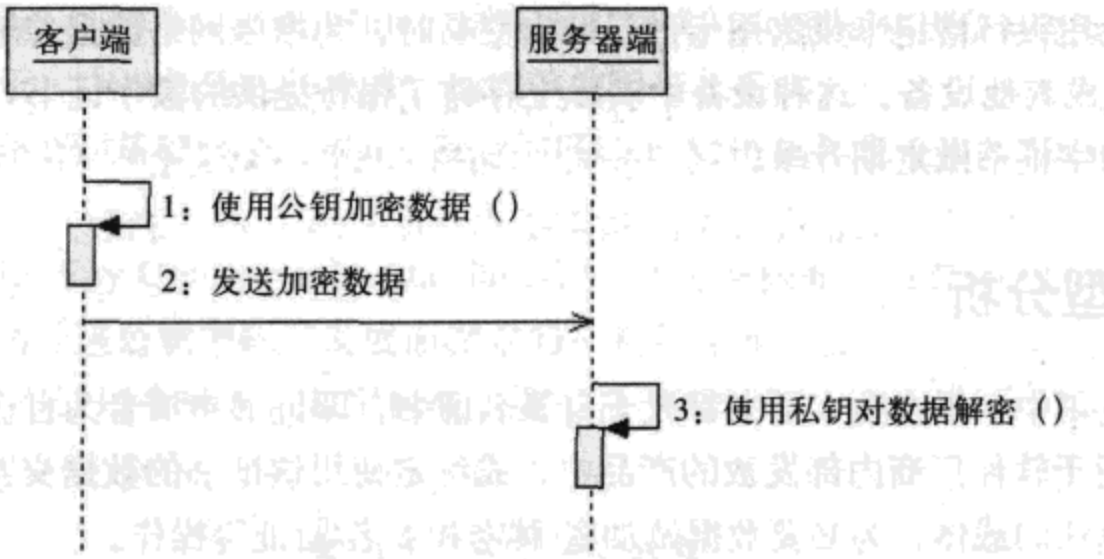


图10-5 客户端请求服务器

客户端请求服务器将按如下步骤进行：

1）由客户端使用公钥对数据加密。

2）由客户端向服务器端发送加密数据。

3）由服务器端使用私钥对数据解密。

读者朋友对这些操作应当不会感到陌生，它与本书之前讲到的基于非对称加密算法的消息传递模型别无二致。

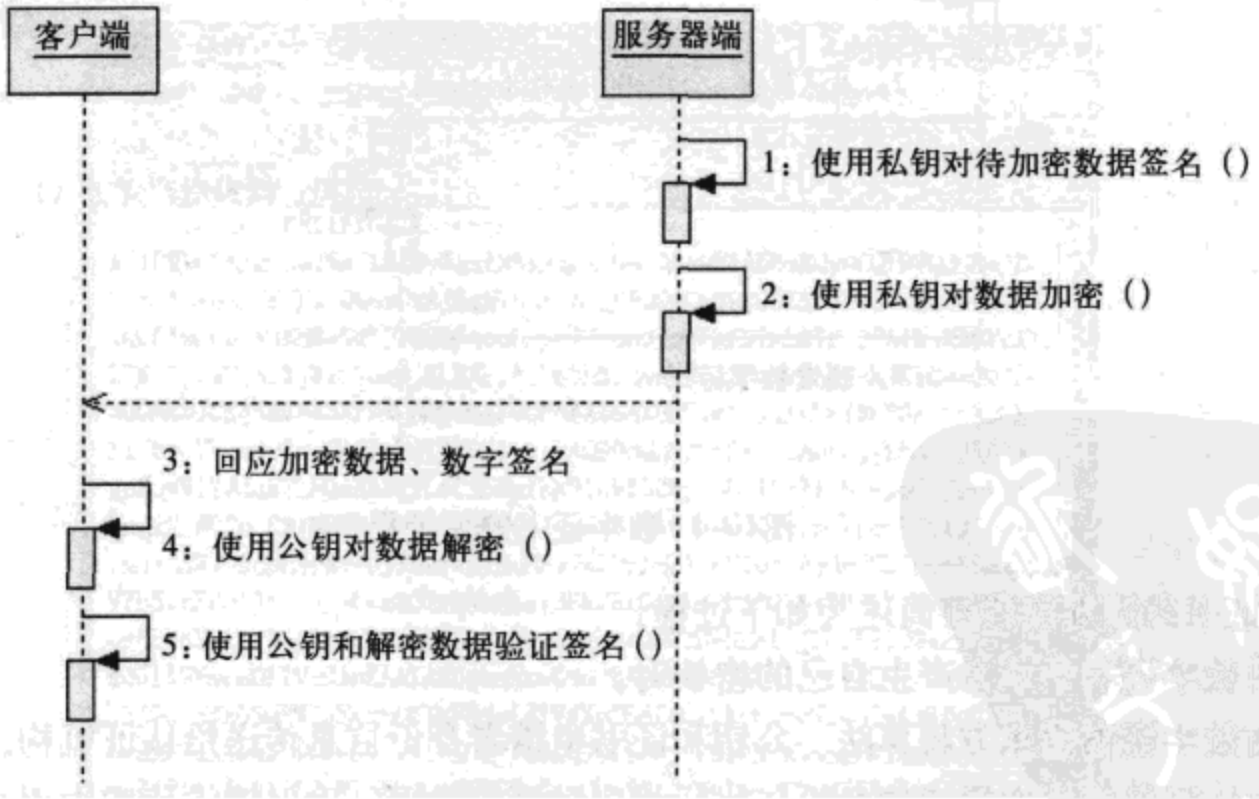


图10-6 服务器端响应客户端

服务器端完成客户端请求处理后，需经过以下几个步骤完成响应：

1）由服务器端使用私钥对待加密数据签名。

2）由服务器端使用私钥对数据加密。

3）由服务器向客户端回应加密数据和数字签名。

更多资源请访问稀酷客(www.ckook.com)

4）由客户端使用公钥对数据解密。

5）由客户端使用公钥和解密数据验证签名。

数字证书的最佳应用环境是在HTTPS安全协议中，使用流程远比上述加密交互流程复杂，但相关操作封装在传输层，对于应用层透明。在HTTPS安全协议中使用非对称加密算法交换密钥，使用对称加密算法对数据进行加密/解密操作，提高加密/解密效率。

10.3 证书管理

任何机构或个人都可以申请数字证书，并使用由CA机构颁发的数字证书为自己的应用保驾护航。

要获得数字证书，我们需要使用数字证书管理工具（如KeyTool和OpenSSL）构建CSR (Certificate Signing Request，数字证书签发申请)，交由CA机构签发，形成最终的数字证书。

本文以Windows操作系统操作KeyTool和OpenSSL为例，演示证书管理的相关操作。获得数字证书后，我们可以使用数字证书提供的算法进行相应实现。

10.3.1 KeyTool证书管理

KeyTool是Java中的数字证书管理工具，用于数字证书的申请、导入、导出和撤销等操作。有关Java 6下的KeyTool命令参见http://java.sun.com/javase/6/docs/technotes/tools/solaris/keytool.html。

KeyTool与本地密钥库相关联，将私钥存于密钥库，公钥则以数字证书输出。KeyTool位于%JDK_HOME%\bin目录中，需要通过命令行进行相应的操作。

1. 构建自签名证书

申请数字证书之前，需要在密钥库中以别名的方式生成本地数字证书，建立相应的加密算法、密钥、有效期等信息，同时需要提供用户身份信息，我们可以为自己签发一个数字证书(即自签名证书)。

这里我们使用“www.zlex.org”作为别名，使用RSA作为加密算法，并规定密钥长度为2048位，使用SHA1withRSA作为数字签名算法，欲签发有效期为36000天的数字证书。完整的命令如代码清单10-1所示。

代码清单10-1 生成本地数字证书1

```
keytool -genkeypair -keyalg RSA -keysize 2048 -sigalg SHA1withRSA -validity
36000 -alias www.zlex.org -keystore zlex.keystore
```

各参数的含义如下所示：

-genkeypair 表示生成密钥。

-keyalg 指定密钥算法，这里指定为RSA算法。

-keysize 指定密钥长度，默认1024位，这里指定为2048位。

-sigalg 指定数字签名算法，这里指定为SHA1withRSA算法。

更多资源请访问稀酷客(www.ckook.com)

-validity　　指定证书有效期，这里指定为36000天。

-alias　　指定别名，这里是www.zlex.org。

-keystore　　指定密钥库存储位置，这里是zlex.keystore。

KeyTool工具支持RSA和DSA共2种算法，且DSA算法为默认算法。

执行上述命令后，可按相应的示输入用户的相关信息，如图10-7所示。

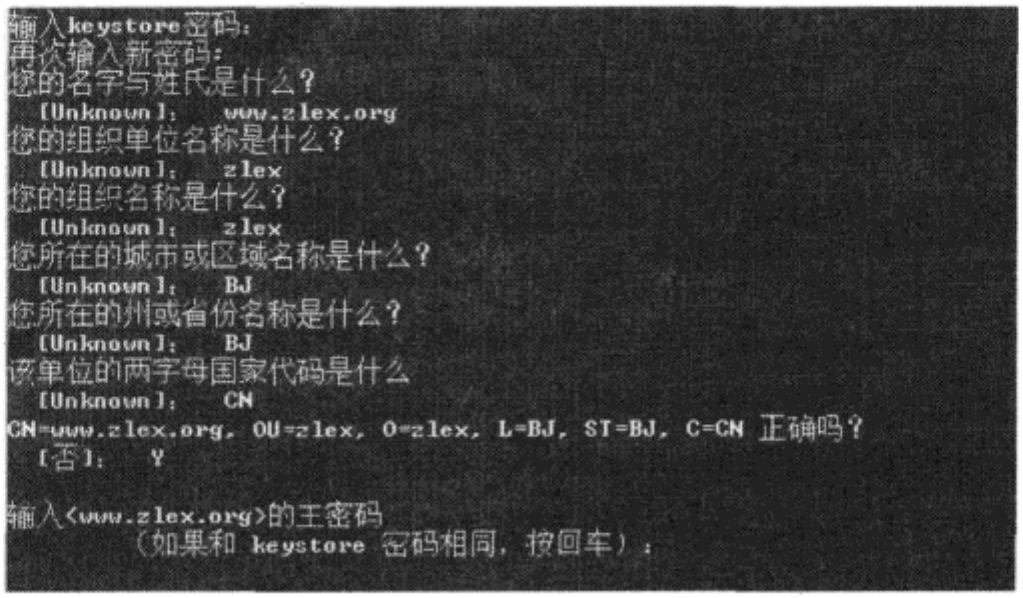


图10-7　生成本地数字证书

这里我们输入密码“123456”，也可通过参数“-storepass”指定密码“123456”。以“www.zlex.org”作为用户的姓名。这里的用户并不是指代现实意义中的用户个体，而是指代网络环境中的用户个体。因此，通常使用域名或带有通配符“*”的泛域名，如“*.zlex.org”标识用户身份。

我们可以使用参数-dname指定用户信息，代替上述手动输入用户信息。完整的命令如代码清单10-2所示。

代码清单10-2　生成本地数字证书2

```
keytool -genkeypair -keyalg RSA -keysize 2048 -sigalg SHA1withRSA -validity
36000 -alias www.zlex.org -keystore zlex.keystore -dname "CN=www.zlex.org, OU=zlex,
O=zlex, L=BJ, ST=BJ, C=CN"
```

经过上述操作后，密钥库中已经创建了数字证书。虽然这时的数字证书并没有经过CA认证，但并不影响我们使用。

我们仍可以将数字证书导出，发送给合作伙伴进行加密交互。完整命令如代码清单10-3所示。

代码清单10-3　导出数字证书

```
keytool -exportcert -alias www.zlex.org -keystore zlex.keystore -file zlex.cer -rfc
```

各参数的含义如下所示：

-exportcert　　表示证书导出操作。

-alias　　指定导别名，这里为www.zlex.org。

-keystore　　指定密钥库文件，这里为zlex.keystore。

更多资源请访问稀酷客(www.ckook.com)

-file　　指定导出文件路径，这里为zlex.cer。

-rfc　　指定以Base64编码格式输出。

这里我们输入密码“123456”，也可通过参数“-storepass”指定密码“123456”，将获得数字证书文件“zlex.cer”。

执行上述命令后，操作结果如图10-8所示。

```
输入keystore密码:
保存在文件中的认证 <zlex.cer>
```

图10-8 导出数字证书

我们可以通过相应的命令在控制台打印数字证书的信息，命令如代码清单10-4所示。

代码清单10-4 打印数字证书

```
keytool -printcert -file zlex.cer
```

控制台打印数字证书信息如图10-9所示。

```
所有者:CN=www.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN
签发人:CN=www.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN
序列号:4b122e83
有效期: Sun Nov 29 16:19:15 CST 2009 至Sat Jun 23 16:19:15 CST 2108
证书指纹:
         MD5:E3:07:50:97:89:BB:04:8D:F2:55:2A:86:32:6E:5A:5F
         SHA1:56:C8:52:C6:6C:BC:31:E0:D6:0D:D4:86:35:CA:31:4A:53:B7:29:DE
         签名算法名称:SHA1withRSA
         版本: 3
```

图10-9 打印数字证书

这里通过KeyTool工具直接导出的证书，是一个自签名的X.509第三版类型的根证书，并以Base64编码保存。自签名证书虽然可以使用，但未经过CA机构认证，几乎没有任何法律效力。

在使用自签名证书时，我们需要将其导入系统。

2. 构建CA签发证书

如果要获取CA机构认证的数字证书，需要将数字证书签发申请（CSR）导出，经由CA机构认证并颁发，同时将认证后的证书导入本地密钥库和信任库。

导出数字证书签发申请操作，如代码清单10-5所示。

代码清单10-5 导出数字证书签发申请

```
keytool -certreq -alias www.zlex.org -keystore zlex.keystore -file zlex.csr -v
```

各参数的含义如下所示：

-certreq　　表示数字证书申请操作。

-alias　　指定别名，这里为www.zlex.org。

-keystore　　指定密钥库文件，这里为zlex.keystore。

-file　　指定导出文件路径，这里为zlex.csr。

-v　　详细信息。

这里我们输入密码“123456”，也可通过参数“-storepass”指定密码“123456”。

更多资源请访问稀酷客(www.ckook.com)

执行上述命令后，将得到一个PKCS#10编码格式的数字证书签发申请文件，即文件“zlex.csr”，操作结果如图10-10所示。



图10-10 导出数字证书签发申请

目前，由VeriSign（http://www.verisign.com/）、GeoTrust（http://www.geotrust.com/）和Thawte（http://www.thawte.com/）国际权威数字证书颁发认证机构“三巨头”签发的数字证书价格不菲，但这三大认证机构几乎都提供了用于测试的数字证书，读者朋友可以提交CSR文件内容，获得相应的签发数字证书。这些用于测试的数字证书通常在时效上扩展功能等方面有限制。

当然，我们可以使用免费的国际权威数字证书颁发认证机构签发数字证书，如CAcert（http://www.cacert.org/）。

获得签发后的数字证书后，需要将其导入信任库。导入数字证书操作，如代码清单10-6所示。

代码清单10-6 导入数字证书

```
keytool -importcert -trustcacerts -alias www.zlex.org -file zlex.cer -keystore zlex.keystore
```

各参数的含义如下所示：

-importcert　　表示导入数字证书。

-trustcacerts　　表示将数字证书导入信任库。

-alias　　指定导别名，这里为www.zlex.org。

-file　　指定导入数字证书文件路径，这里为zlex.cer。

-keystore　　指定密钥库文件，这里为zlex.keystore。

这里我们输入密码“123456”，也可通过参数“-storepass”指定密码“123456”，并输入“Y”确认导入该证书。本文以自签名证书导入为例，执行上述命令操作的结果如图10-11所示。

导入证书后，我们可以通过相关命令查看该证书，命令如代码清单10-7所示。

代码清单10-7 查看导入数字证书

```
keytool -list -alias www.zlex.org -keystore zlex.keystore
```

各参数的含义如下所示：

-list　　表示导入数字证书。

-alias　　指定别名，这里为www.zlex.org。

-keystore　　指定密钥库文件，这里为zlex.keystore。

这里我们输入密码“123456”，也可通过参数“-storepass”指定密码“123456”。

执行上述命令操作的结果如图10-12所示。

我们也可以加入参数“-v”或“-rfc”查看该证书的详细信息。

完成上述操作后，我们需要使用证书导出命令导出数字证书（见代码清单10-3），并将其发放给合作伙伴使用。

更多资源请访问稀酷客(www.ckook.com)

```
输入keystore密码:
再次输入新密码:
所有者:CN=www.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN
签发人:CN=www.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN
序列号:4b122e83
有效期: Sun Nov 29 16:19:15 CST 2009 至Sat Nov 23 16:19:15 CST 2018
证书指纹:
         MD5:E3:07:50:97:89:BB:04:8D:F2:55:2A:86:32:6E:5A:5F
         SHA1:56:C8:52:C6:6C:BC:31:E0:D6:0D:D4:86:35:CA:31:4A:53:B7:29:DE
         签名算法名称:SHA1withRSA
         版本: 3

扩展:

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 90 99 54 A2 43 7F 9C 5A   CC 7A 1A A1 ED 53 67 EF  ..T.C..Z.z...Sg.
0010: 87 AA 71 18                                        ..q.
]
]

#2: ObjectId: 2.5.29.19 Criticality=false
BasicConstraints:[
  CA:true
  PathLen:2147483647
]

#3: ObjectId: 2.5.29.35 Criticality=false
AuthorityKeyIdentifier [
KeyIdentifier [
0000: 90 99 54 A2 43 7F 9C 5A   CC 7A 1A A1 ED 53 67 EF  ..T.C..Z.z...Sg.
0010: 87 AA 71 18                                        ..q.
]

[CN=www.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN]
SerialNumber: [    4b122e83]
]

信任这个认证? [否]:  Y
认证已添加至keystore中
```

图10-11 导入数字证书

```
输入keystore密码:
www.zlex.org, 2009-11-29, PrivateKeyEntry,
认证指纹 (MD5):  E3:07:50:97:89:BB:04:8D:F2:55:2A:86:32:6E:5A:5F
```

图10-12 查看导入数字证书

10.3.2 OpenSSL证书管理

OpenSSL（http://www.openssl.org/）是一个开放源代码软件包，由Eric A. Young 和 Tim J. Hudson等人编写，实现了SSL及相关加密技术，是最常用的证书管理工具。OpenSSL功能远胜于KeyTool，可用于根证书、服务器证书和客户证书的管理。相关操作可以参考官方文档（http://www.openssl.org/docs/apps/openssl.html）。

我们可以在OpenSSL官网下载页面（http://www.openssl.org/source/）下载最新的源码（目前最新版本为0.9.8l），下载后需对源码包进行编译后方能使用。OpenSSL官网提供了Windows版的二进制发行版地址（http://www.slproweb.com/products/Win32OpenSSL.html），我们可以在该页面下载最新的Windows版OpenSSL。

本文以Windows版OpenSSL为例，演示如何构建数字证书。

更多资源请访问稀酷客(www.ckook.com)

1. 准备工作

Windows版OpenSSL分为Win32和Win64共2个平台版本，请读者朋友注意选择合适的发行版，本文选用Win32 OpenSSL v0.9.8l Light演示数字证书构建相关操作。

有关Windows版OpenSSL安装操作与一般软件安装并无差别，需要读者朋友注意的是OpenSSL的相关配置。

❑ 环境变量

为了在命令行下方便使用OpenSSL，我们需要在Windows环境变量对话框中对OpenSSL进行设置，如图10-13所示。

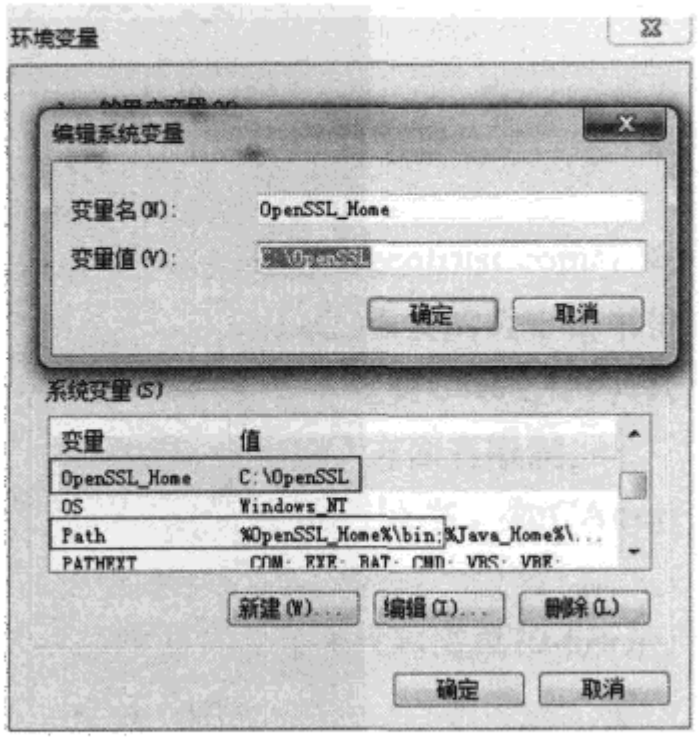


图10-13 OpenSSL系统变量配置

设置系统变量OpenSSL_Home，并将其指向OpenSSL的安装目录（C:\OpenSSL）。同时，将其执行目录（%OpenSSL_Home%\bin;）加入系统变量Path中。完成上述操作后，在命令行下执行OpenSSL命令，如图10-14所示。



图10-14 OpenSSL命令行操作

请读者朋友注意，如果使用Windows Vista或Windows 7操作系统使用OpenSSL相关命令时，必须使用管理员身份。

❑ 工作目录

打开OpenSSL配置文件openssl.cfg（%OpenSSL_Home%\bin\openssl.cfg），找到配置[CA_default]，如图10-15所示。

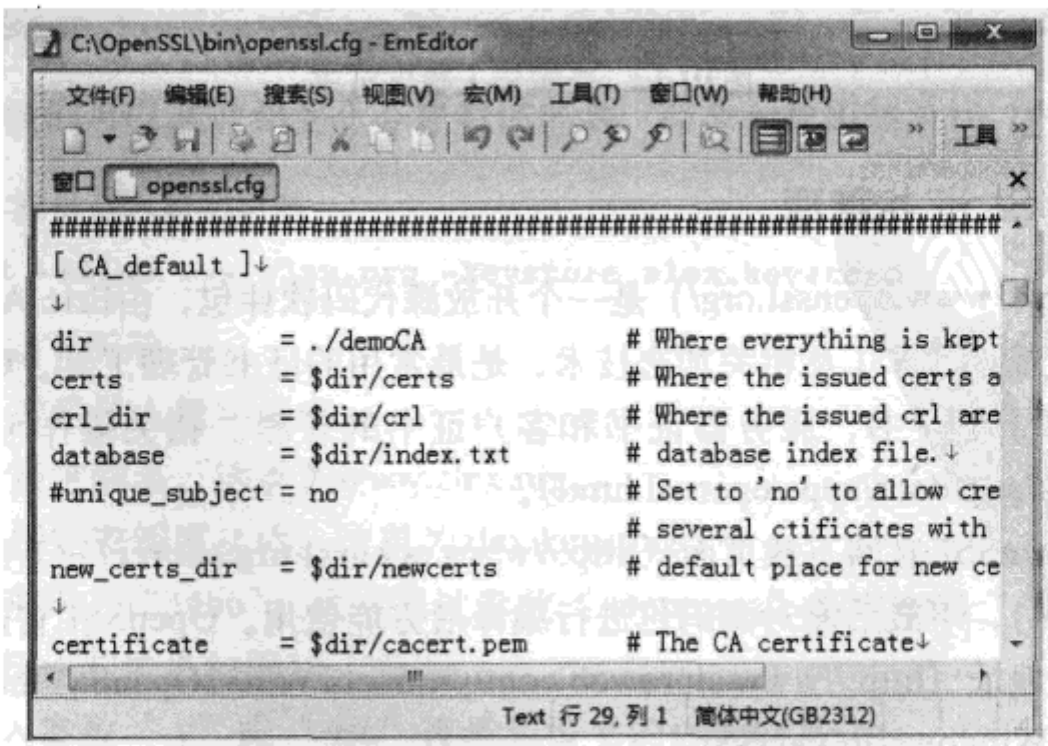


图10-15 OpenSSL初始配置

注意变量dir，它指向的是CA工作目录，本文将路径D:/ca作为CA工作目录，对变量dir做相

更多资源请访问稀酷客(www.ckook.com)

应修改。对于其他变量，我们无需修改。

建立CA工作目录后，我们需要构建一些子目录，用于存放证书、密钥等。完整命令如代码清单10-8所示。

代码清单10-8 构建CA子目录

```
echo 构建已发行证书存放目录 certs
mkdir certs
echo 构建新证书存放目录 newcerts
mkdir newcerts
echo 构建私钥存放目录 private
mkdir private
echo 构建证书吊销列表存放目录 crl
mkdir crl
```

我们将在创建证书时用到上述目录，最终在certs目录中获得证书文件。

接下来，我们需要构建一些文件，完整命令如代码清单10-9所示。

代码清单10-9 构建相关文件

```
echo 构建索引文件 index.txt
echo 0>index.txt
echo 构建序列号文件 serial
echo 01>serial
```

完成上述操作后，我们就可以进行证书的构建和签发工作了。

2. 构建根证书

在构建根证书前，需要构建随机数文件（.rand），完整命令如代码清单10-10所示。

代码清单10-10 构建随机数

```
echo 构建随机数 private/.rand
openssl rand -out private/.rand 1000
```

各参数的含义如下所示：

rand　　随机数命令。

-out　　输出文件路径，这里将随机数文件输出到private目录下。

这里的参数1000，指用来产生伪随机字节数。

上述命令执行结果如图10-16所示。

```
构建随机数 private/.rand
Loading 'screen' into random state - done
```

图10-16 构建随机数文件

请读者朋友注意，如果在Windows Vista或Windows 7中未能使用管理权限执行上述命令，将无法创建随机数文件，如图10-17所示。

```
构建随机数 private/.rand
Loading 'screen' into random state - done
unable to write 'random state'
```

图10-17 构建随机数文件失败

更多资源请访问稀酷客(www.ckook.com)

OpenSSL通常使用PEM（Privacy Enbanced Mail，隐私增强邮件）编码格式保存私钥。

接下来，我们需要构建根证书密钥（ca.key.pem），完整命令如代码清单10-11所示。

代码清单10-11 构建根证书私钥

```
echo 构建根证书私钥 private/ca.key.pem
openssl genrsa -aes256 -out private/ca.key.pem 2048
```

各参数的含义如下所示：

genrsa　　产生RSA密钥命令。

-aes256　　使用AES算法（256位密钥）对产生的私钥加密。可选算法包括DES、DESede、IDEA和AES。

-out　　输出路径，这里指private/ca.key.pem。

这里的参数2048，指RSA密钥长度位数，默认长度为512位。

上述命令执行结果，如图10-18所示。

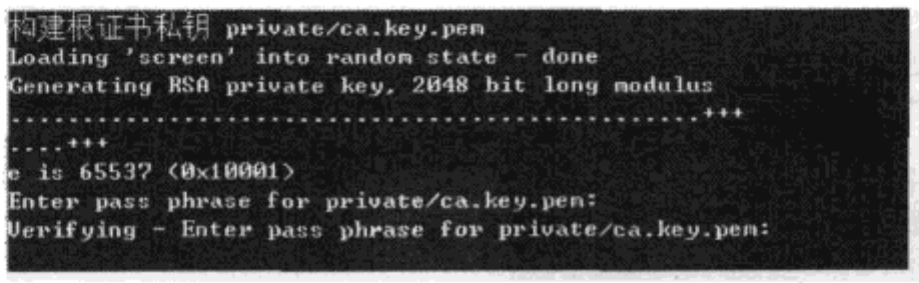


图10-18 构建根证书私钥

这时我们需要输入根证书密码“123456”。

完成密钥构建操作后，我们需要生成根证书签发申请文件（ca.csr），完整命令如代码清单10-12所示。

代码清单10-12 生成根证书签发申请

```
echo 生成根证书签发申请 private/ca.csr
openssl req -new -key private/ca.key.pem -out private/ca.csr -subj
"/C=CN/ST=BJ/L=BJ/O=zlex/OU=zlex/CN=*.zlex.org"
```

各参数的含义如下所示：

req　　产生证书签发申请命令。

-new　　表示新请求。

-key　　密钥，这里为private/ca.key.pem文件。

-out　　输出路径，这里为private/ca.csr文件。

-subj　　指定用户信息，这里使用泛域名“*.zlex.org”作为用户名。

上述命令执行结果，如图10-19所示。

这时我们需要输入根证书密码“123456”。

得到根证书签发申请文件后，我们可以将其发送给CA机构签发。当然，我们也可以自行签发根证书。签发根证书完整命令如代码清单10-13所示。

更多资源请访问稀酷客(www.ckook.com)

```
生成根证书签发申请 private/ca.csr
Enter pass phrase for private/ca.key.pem:
Loading 'screen' into random state - done
```

图10-19 生成根证书签发申请

代码清单10-13 签发根证书

```
echo 签发根证书 private/ca.cer
openssl x509 -req -days 10000 -sha1 -extensions v3_ca -signkey
private/ca.key.pem -in private/ca.csr -out certs/ca.cer
```

各参数的含义如下所示：

x509	签发X.509格式证书命令。
-req	表示证书输入请求。
-days	表示有效天数，这里为10000天。
-sha1	表示证书摘要算法，这里为SHA1算法。
-extensions	表示按OpenSSL配置文件v3_ca项添加扩展。
-signkey	表示自签名密钥，这里为private/ca.key.pem。
-in	表示输入文件，这里为private/ca.csr。
-out	表示输出文件，这里为certs/ca.cer。

上述命令执行结果，如图10-20所示。

```
签发根证书 private/ca.cer
Loading 'screen' into random state - done
Signature ok
subject=/C=CN/ST=BJ/L=BJ/O=zlex/OU=zlex/CN=*.zlex.org
Getting Private key
Enter pass phrase for private/ca.key.pem:
```

图10-20 签发根证书

这时我们需要输入根证书密码“123456”。

OpenSSL产生的数字证书不能在Java语言环境中直接使用，需要将其转化为PKCS#12编码格式。完整命令如代码清单10-14所示。

代码清单10-14 根证书转换

```
echo 根证书转换 private/ca.p12
openssl pkcs12 -export -cacerts -inkey private/ca.key.pem -in certs/ca.cer -out certs/ca.p12
```

各参数的含义如下所示：

pkcs12	PKCS#12编码格式证书命令。
-export	表示导出证书。
-cacerts	表示仅导出CA证书。
-inkey	表示输入密钥，这里为private/ca.key.pem。
-in	表示输入文件，这里为certs/ca.cer。
-out	表示输出文件，这里为certs/ca.p12。

上述命令执行结果，如图10-21所示。

更多资源请访问稀酷客(www.ckook.com)

```
根证书转换 private/ca.p12
Loading 'screen' into random state - done
Enter pass phrase for private/ca.key.pem:
Enter Export Password:
Verifying - Enter Export Password:
```

图10-21 根证书转换

这时我们需要输入根证书密码“123456”。

个人信息交换文件（PKCS#12）可以作为密钥库或信任库使用，我们可以通过KeyTool查看该密钥库的详细信息。完整命令如代码清单10-15所示。

代码清单10-15 查看密钥库信息

```
keytool -list -keystore certs/ca.p12 -storetype pkcs12 -v -storepass 123456
```

注意，这里参数-storetype值为“pkcs12”。上述命令执行结果如图10-22所示。

```
Keystore 类型: PKCS12
Keystore 提供者: SunJSSE

您的 keystore 包含 1 输入

别名名称: 1
创建日期: 2009-12-13
项类型: PrivateKeyEntry
认证链长度: 1
认证 [1]:
所有者:CN=www.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN
签发人:CN=*.zlex.org, OU=zlex, O=zlex, L=BJ, ST=BJ, C=CN
序列号:9d7182b4f37e3306
有效期: Tue Dec 13 10:48:02 CST 2009 至Fri Dec 11 10:48:02 CST 2019
证书指纹:
        MD5:48:D6:F8:1B:9B:43:D9:E5:76:2A:31:21:5A:DA:46:10
        SHA1:12:7D:77:F5:A1:EC:64:28:27:E4:DE:D2:D3:B1:DB:1A:BD:01:A7:20
        签名算法名称:SHA1withRSA
        版本: 1

*******************************************
*******************************************
```

图10-22 查看密钥库信息

现在，我们已经构建了根证书（ca.cer），我们可以使用根证书签发服务器证书和客户证书。

3. 构建服务器证书

服务器证书的构建与根证书构建相似，首先需要构建私钥。完整命令如代码清单10-16所示。

代码清单10-16 构建服务器私钥

```
echo 构建服务器私钥 private/server.key.pem
openssl genrsa -aes256 -out private/server.key.pem 2048
```

各参数的含义如下所示：

genrsa　　产生RSA密钥命令。

-aes256　　使用AES算法（256位密钥）对产生的私钥加密。可选算法包括DES、DESede、IDEA和AES。

-out　　输出路径，这里指private/server.key.pem。

更多资源请访问稀酷客(www.ckook.com)

这里的参数2048，指RSA密钥长度位数，默认长度为512位。

上述命令执行结果，如图10-23所示。

```
构建服务器证书私钥 private/server.key.pem
Loading 'screen' into random state - done
Generating RSA private key, 2048 bit long modulus
.....................................................+++
.....................................................+++
e is 65537 (0x10001)
Enter pass phrase for private/server.key.pem:
Verifying - Enter pass phrase for private/server.key.pem:
```

图10-23 产生服务器证书密钥

这时我们需要输入服务器证书密码“123456”。

完成服务器证书密钥构建后，我们需要产生服务器证书签发申请。完整命令如代码清单10-17所示。

代码清单10-17 生成服务器证书签发申请

```
echo 生成服务器证书签发申请 private/server.csr
openssl req -new -key private/server.key.pem -out private/server.csr -subj
"/C=CN/ST=BJ/L=BJ/O=zlex/OU=zlex/CN=www.zlex.org"
```

各参数的含义如下所示：

req　　产生证书签发申请命令。

-new　　表示新请求。

-key　　密钥，这里为private/ca.key.pem文件。

-out　　输出路径，这里为private/ca.csr文件。

-subj　　指定用户信息，这里使用域名“www.zlex.org”作为用户名。

上述命令执行结果，如图10-24所示。

```
生成服务器证书签发申请 private/server.csr
Enter pass phrase for private/server.key.pem:
Loading 'screen' into random state - done
```

图10-24 生成服务器证书签发申请

这时我们需要输入服务器证书密码“123456”。

我们已经获得了根证书，可以使用根证书签发服务器证书。完整命令如代码清单10-18所示。

代码清单10-18 签发服务器证书

```
echo 签发服务器证书 private/server.cer
openssl x509 -req -days 3650 -sha1 -extensions v3_req -CA certs/ca.cer -CAkey
private/ca.key.pem -CAserial ca.srl -CAcreateserial -in private/server.csr -out
certs/server.cer
```

各参数的含义如下所示：

x509　　签发X.509格式证书命令。

-req　　表示证书输入请求。

更多资源请访问稀酷客(www.ckook.com)

-days	表示有效天数，这里为3650天。
-sha1	表示证书摘要算法，这里为SHA1算法。
-extensions	表示按OpenSSL配置文件v3_req项添加扩展。
-CA	表示CA证书，这里为certs/ca.cer。
-CAkey	表示CA证书密钥，这里为private/ca.key.pem。
-CAserial	表示CA证书序列号文件，这里为ca.srl。
-CAcreateserial	表示创建CA证书序列号。
-in	表示输入文件，这里为private/server.csr。
-out	表示输出文件，这里为certs/server.cer。

上述命令执行结果，如图10-25所示。

```
签发服务器证书 private/server.cer
Loading 'screen' into random state - done
Signature ok
subject=/C=CN/ST=BJ/L=BJ/O=zlex/OU=zlex/CN=www.zlex.org
Getting CA Private Key
Enter pass phrase for private/ca.key.pem:
```

图10-25 签发服务器证书

这时我们需要输入服务器证书密码“123456”。'

这里我们同样需要将OpenSSL产生的数字证书转化为PKCS#12编码格式。完整命令如代码清单10-19所示。

代码清单10-19 服务器证书转换

```
echo 服务器证书转换 private/server.p12
openssl pkcs12 -export -clcerts -inkey private/server.key.pem -in
certs/server.cer -out certs/server.p12
```

各参数的含义如下所示：

pkcs12	PKCS#12编码格式证书命令。
-export	表示导出证书。
-clcerts	表示仅导出客户证书。
-inkey	表示输入密钥，这里为private/server.key.pem。
-in	表示输入文件，这里为certs/ca.cer。
-out	表示输出文件，这里为certs/server.p12。

上述命令执行结果，如图10-26所示。

```
服务器证书转换 private/server.p12
Loading 'screen' into random state - done
Enter pass phrase for private/server.key.pem:
Enter Export Password:
Verifying - Enter Export Password:
```

图10-26 服务器证书转换

这时我们需要输入服务器证书密码“123456”。

现在，我们已经构建了服务器证书（server.cer），并可使用该证书构建基于单向认证的网

更多资源请访问稀酷客(www.ckook.com)

络交互平台。

4. 构建客户证书

客户证书的构建与服务器证书构建基本一致，首先需要构建私钥。完整命令如代码清单10-20所示。

代码清单10-20 产生客户私钥

```
echo 产生客户私钥 private/client.key.pem
openssl genrsa -aes256 -out private/client.key.pem 2048
```

各参数的含义如下所示：

genrsa　　产生RSA密钥命令。

-aes256　　使用AES算法（256位密钥）对产生的私钥加密。可选算法包括DES、DESede、IDEA和AES。

-out　　输出路径，这里指private/client.key.pem。

这里的参数2048，指RSA密钥长度位数，默认长度为512位。

上述命令执行结果，如图10-27所示。

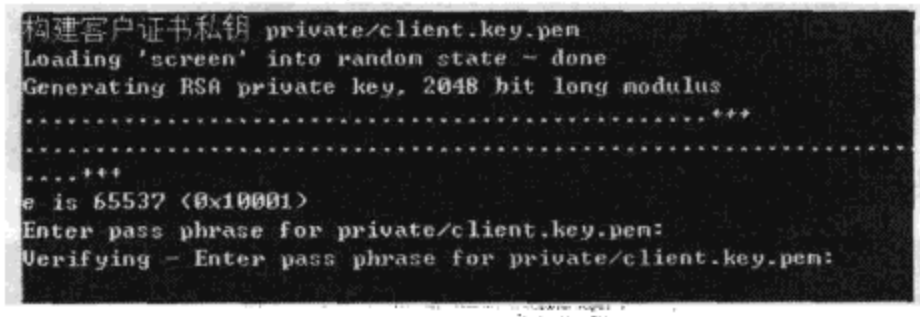


图10-27 产生客户密钥

这时我们需要输入客户证书密码“123456”。

完成客户证书密钥构建后，我们需要产生客户证书签发申请。完整命令如代码清单10-21所示。

代码清单10-21 生成客户证书签发申请

```
echo 生成客户证书签发申请 client.csr
openssl req -new -key private/client.key.pem -out private/client.csr -subj
"/C=CN/ST=BJ/L=BJ/O=zlex/OU=zlex/CN=zlex"
```

各参数的含义如下所示：

req　　产生证书签发申请命令。

-new　　表示新请求。

-key　　密钥，这里为private/client.key.pem文件。

-out　　输出路径，这里为private/client.csr文件。

-subj　　指定用户信息，这里使用“zlex”作为用户名。

上述命令执行结果，如图10-28所示。

更多资源请访问稀酷客(www.ckook.com)

```
生成客户证书签发申请 private/client.csr
Enter pass phrase for private/client.key.pem:
Loading 'screen' into random state - done
```

图10-28 生成客户证书签发申请

这时我们需要输入客户证书密码“123456”。

我们已经获得了根证书，可以使用根证书签发客户证书（client.cer）。完整命令如代码清单10-22所示。

代码清单10-22 签发客户证书

```
echo 签发客户证书 certs/client.cer
openssl ca -days 3650 -in private/client.csr -out certs/client.cer -cert
certs/ca.cer -keyfile private/ca.key.pem
```

各参数的含义如下所示：

ca	签发证书命令。
-days	表示证书有效期，这里为3650天。
-in	表示输入文件，这里为private/client.csr。
-out	表示输出文件，这里为certs/server.cer。
-cert	表示证书文件，这里为certs/ca.cer。
-keyfile	表示根证书密钥文件，这里为private/ca.key.pem。

上述命令执行结果，如图10-29所示。

```
签发客户证书 certs/client.cer
Using configuration from C:\OpenSSL\bin\openssl.cfg
Loading 'screen' into random state - done
Enter pass phrase for private/ca.key.pem:
Check that the request matches the signature
Signature ok
Certificate Details:
        Serial Number: 1 (0x1)
        Validity
            Not Before: Dec 13 11:10:38 2009 GMT
            Not After : Dec 11 11:10:38 2019 GMT
        Subject:
            countryName               = CN
            stateOrProvinceName       = BJ
            organizationName          = zlex
            organizationalUnitName    = zlex
            commonName                = zlex
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            Netscape Comment:
                OpenSSL Generated Certificate
            X509v3 Subject Key Identifier:
                C1:11:37:70:5F:70:03:BC:09:39:4F:BF:2D:41:9E:9A:76:9A:13:C3
            X509v3 Authority Key Identifier:
                DirName:/C=CN/ST=BJ/L=BJ/O=zlex/OU=zlex/CN=*.zlex.org
                serial:C1:A7:43:35:FD:9C:A8:A9

Certificate is to be certified until Dec 11 11:10:38 2019 GMT (3650 days)
Sign the certificate? [y/n]:y


1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

图10-29 签发客户证书

更多资源请访问稀酷客(www.ckook.com)

这时我们需要输入客户证书密码“123456”，并同意签发证书。

最后，我们需要将获得客户证书转化Java语言可以识别的PKCS#12编码格式。完整命令如代码清单10-23所示。

代码清单10-23 客户证书转换

```
echo 客户证书转换 certs/client.p12
openssl pkcs12 -export -inkey private/client.key.pem -in certs/client.cer -out certs/client.p12
```

各参数的含义如下所示：

pkcs12　　PKCS#12编码格式证书命令。

-export　　表示导出证书。

-clcerts　　表示仅导出客户证书。

-inkey　　表示输入密钥，这里为private/client.key.pem。

-in　　表示输入文件，这里为certs/client.cer。

-out　　表示输出文件，这里为certs/client.p12。

上述命令执行结果，如图10-30所示。

```
客户证书转换 certs/client.p12
Loading 'screen' into random state - done
Enter pass phrase for private/client.key.pem:
Enter Export Password:
Verifying - Enter Export Password:
```

图10-30 客户证书转换

这时我们需要输入客户证书密码“123456”。

至此，我们完成了双向认证的所需的全部证书。我们将在后续章节中介绍如何构建基于双向认证的网络应用。

10.4 证书使用

Java 6提供了完善的数字证书管理实现，我们几乎无需关注相关具体算法，仅通过操作密钥库和数字证书就可完成相应的加密/解密和签名/验证操作。密钥库管理私钥，数字证书管理公钥，私钥和密钥分属消息传递两方，进行加密消息传递。

因此，我们可以将密钥库看做私钥相关操作的入口，数字证书则是公钥相关操作的入口。

本文以KeyTool产生的密钥库和证书为例，演示证书使用相关操作。加载密钥库需要提供密钥库文件路径和密钥库密码，如代码清单10-24所示。

代码清单10-24 加载密钥库

```
/**
 * 获得KeyStore
 * @param keyStorePath 密钥库路径
* @param password 密码
 * @return KeyStore 密钥库
```

更多资源请访问稀酷客(www.ckook.com)

```
     */
    private static KeyStore getKeyStore(String keyStorePath, String password) throws Exception {
        // 实例化密钥库
        KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());
        // 获得密钥库文件流
        FileInputStream is = new FileInputStream(keyStorePath);
        // 加载密钥库
        ks.load(is, password.toCharArray());
        // 关闭密钥库文件流
        is.close();
        return ks;
    }
```

加载密钥库后，我们就能通过相应的方法获得私钥，也可以获得数字证书。获得私钥实现如代码清单10-25所示。

代码清单10-25　由密钥库获得私钥

```
// 获得密钥库
KeyStore ks = getKeyStore(keyStorePath, password);
// 获得私钥
PrivateKey privateKey = (PrivateKey) ks.getKey(alias, password.toCharArray());
```

这里通过我们已实现的getKeyStore()获得密钥库，输入别名（参数alias）和密码（参数password）即可获得私钥。获得私钥后，可按照本书第8章介绍的RSA算法实现进行“私钥加密，公钥解密”和“公钥加密，私钥解密”两项操作。

如果我们需要从密钥库中获得签名算法，只能通过由密钥库中获取的数字证书并强转为X509Certificate实例，通过其getSigAlgName()方法获得对应的签名算法。完整代码如代码清单10-26所示。

代码清单10-26　由密钥库获得数字证书构建数字签名对象

```
// 获得密钥库
KeyStore ks = getKeyStore(keyStorePath, password);
// 获得证书
X509Certificate x509Certificate = (X509Certificate) ks.getCertificate(alias);
// 构建签名，由证书指定签名算法
Signature signature = Signature.getInstance(x509Certificate.getSigAlgName());
```

获得数字签名对象后，我们可使用私钥进行签名操作。相关实现与本书第9章RSA算法实现非常相似。

相比于密钥库操作，数字证书的操作更为简单，我们只需要给出数字整证书的路径，并加载它即可。完整代码实现如代码清单10-27所示。

代码清单10-27　加载数字证书

```
/**
 * 获得Certificate
```

更多资源请访问稀酷客(www.ckook.com)

```
 * @param certificatePath 证书路径
 * @return Certificate 证书
 * @throws Exception
 */
private static Certificate getCertificate(String certificatePath) throws Exception {
    // 实例化证书工厂
    CertificateFactory certificateFactory = CertificateFactory.getInstance("X.509");
    // 取得证书文件流
    FileInputStream in = new FileInputStream(certificatePath);
    // 生成证书
    Certificate certificate = certificateFactory.generateCertificate(in);
    // 关闭证书文件流
    in.close();
    return certificate;
}
```

目前，Java 6仅支持X.509类型的数字证书。

通过以上方式加载得到数字证书后，我们可以直接获取公钥，如代码清单10-28所示。

代码清单10-28 由数字证书获得公钥

```
// 获得证书
Certificate certificate = getCertificate(certificatePath);
// 获得公钥
PublicKey publicKey = certificate.getPublicKey();
```

得到公钥后，我们就可以参考本书第8章介绍的RSA算法实现进行“公钥加密，私钥解密”和“私钥加密，公钥解密”两项操作。

如果使用数字证书进行验证签名操作时，需要将获得的证书对象强转为X509Certificate实例。完整代码如代码清单10-29所示。

代码清单10-29 初始化签名对象

```
// 获得证书
X509Certificate x509Certificate = (X509Certificate) getCertificate(certificatePath);
// 由证书构建签名
Signature signature = Signature.getInstance(x509Certificate.getSigAlgName());
// 由证书初始化签名，实际上是使用了证书中的公钥
signature.initVerify(x509Certificate);
```

这里需要注意的是，数字签名对象初始化（initVerify()方法）时使用了数字证书而非公钥。该方法在内部实际上是使用数字证书的公钥。

至此，我们可使用数字证书做“私钥验证”操作。

上述操作的完整代码实现如代码清单10-30所示。

代码清单10-30 基于密钥库和数字证书的加密/解密和签名/验证操作

```
import java.io.FileInputStream;
import java.security.KeyStore;
```

更多资源请访问稀酷客(www.ckook.com)

```
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.Signature;
import java.security.cert.Certificate;
import java.security.cert.CertificateFactory;
import java.security.cert.X509Certificate;
import javax.crypto.Cipher;
/**
 * 证书组件
 * @author 梁栋
 * @version 1.0
 */
public abstract class CertificateCoder {
    // 类型证书X509
    public static final String CERT_TYPE = "X.509";
    /**
     * 由KeyStore获得私钥
     * @param keyStorePath 密钥库路径
     * @param alias 别名
     * @param password 密码
     * @return PrivateKey 私钥
     * @throws Exception
     */
    private static PrivateKey getPrivateKeyByKeyStore(String keyStorePath,
    String alias, String password) throws Exception {
        // 获得密钥库
        KeyStore ks = getKeyStore(keyStorePath, password);
        // 获得私钥
        return (PrivateKey) ks.getKey(alias, password.toCharArray());
    }
    /**
     * 由Certificate获得公钥
     * @param certificatePath 证书路径
     * @return PublicKey 公钥
     * @throws Exception
     */
    private static PublicKey getPublicKeyByCertificate(String certificatePath)
    throws Exception {
        // 获得证书
        Certificate certificate = getCertificate(certificatePath);
        // 获得公钥
        return certificate.getPublicKey();
    }
    /**
     * 获得Certificate
     * @param certificatePath 证书路径
     * @return Certificate 证书
```

更多资源请访问稀酷客(www.ckook.com)

```
     * @throws Exception
     */
    private static Certificate getCertificate(String certificatePath) throws Exception {
            // 实例化证书工厂
            CertificateFactory certificateFactory = CertificateFactory.getInstance (CERT_TYPE);
            // 取得证书文件流
            FileInputStream in = new FileInputStream(certificatePath);
            // 生成证书
            Certificate certificate = certificateFactory.generateCertificate(in);
            // 关闭证书文件流
            in.close();
            return certificate;
    }
    /**
     * 获得Certificate
     * @param keyStorePath 密钥库路径
     * @param alias 别名
     * @param password 密码
     * @return Certificate 证书
     * @throws Exception
     */
    private static Certificate getCertificate(String keyStorePath, String alias,
    String password) throws Exception {
            // 获得密钥库
            KeyStore ks = getKeyStore(keyStorePath, password);
            // 获得证书
            return ks.getCertificate(alias);
    }
    /**
     * 获得KeyStore
     * @param keyStorePath 密钥库路径
     * @param password 密码
     * @return KeyStore 密钥库
     * @throws Exception
     */
    private static KeyStore getKeyStore(String keyStorePath, String password)
    throws Exception {
            // 实例化密钥库
            KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());
            // 获得密钥库文件流
            FileInputStream is = new FileInputStream(keyStorePath);
            // 加载密钥库
            ks.load(is, password.toCharArray());
            // 关闭密钥库文件流
            is.close();
            return ks;
    }
    /**
```

更多资源请访问稀酷客(www.ckook.com)

```
     * 私钥加密
     * @param data 待加密数据
     * @param keyStorePath 密钥库路径
     * @param alias 别名
     * @param password 密码
     * @return byte[] 加密数据
     * @throws Exception
     */
    public static byte[] encryptByPrivateKey(byte[] data, String keyStorePath,
    String alias, String password) throws Exception {
            // 取得私钥
            PrivateKey privateKey = getPrivateKeyByKeyStore(keyStorePath, alias, password);
            // 对数据加密
            Cipher cipher = Cipher.getInstance(privateKey.getAlgorithm());
            cipher.init(Cipher.ENCRYPT_MODE, privateKey);
            return cipher.doFinal(data);
    }
    /**
     * 私钥解密
     * @param data 待解密数据
     * @param keyStorePath 密钥库路径
     * @param alias 别名
     * @param password 密码
     * @return byte[] 解密数据
     * @throws Exception
     */
    public static byte[] decryptByPrivateKey(byte[] data, String keyStorePath,
    String alias, String password) throws Exception {
            // 取得私钥
            PrivateKey privateKey = getPrivateKeyByKeyStore(keyStorePath, alias, password);
            // 对数据加密
            Cipher cipher = Cipher.getInstance(privateKey.getAlgorithm());
            cipher.init(Cipher.DECRYPT_MODE, privateKey);
            return cipher.doFinal(data);
    }
    /**
     * 公钥加密
     * @param data 待加密数据
     * @param certificatePath 证书路径
     * @return byte[] 加密数据
     * @throws Exception
     */
    public static byte[] encryptByPublicKey(byte[] data, String certificatePath)
    throws Exception {
            // 取得公钥
            PublicKey publicKey = getPublicKeyByCertificate(certificatePath);
            // 对数据加密
            Cipher cipher = Cipher.getInstance(publicKey.getAlgorithm());
```

更多资源请访问稀酷客(www.ckook.com)

```
        cipher.init(Cipher.ENCRYPT_MODE, publicKey);
        return cipher.doFinal(data);
    }
    /**
     * 公钥解密
     * @param data 待解密数据
     * @param certificatePath 证书路径
     * @return byte[] 解密数据
     * @throws Exception
     */
    public static byte[] decryptByPublicKey(byte[] data, String certificatePath)
    throws Exception {
        // 取得公钥
        PublicKey publicKey = getPublicKeyByCertificate(certificatePath);
        // 对数据加密
        Cipher cipher = Cipher.getInstance(publicKey.getAlgorithm());
        cipher.init(Cipher.DECRYPT_MODE, publicKey);
        return cipher.doFinal(data);
    }
    /**
     * 签名
     * @param keyStorePath 密钥库路径
     * @param alias 别名
     * @param password 密码
     * @return byte[] 签名
     * @throws Exception
     */
    public static byte[] sign(byte[] sign, String keyStorePath, String alias,
    String password) throws Exception {
        // 获得证书
        X509Certificate x509Certificate = (X509Certificate) getCertificate
        (keyStorePath, alias, password);
        // 构建签名，由证书指定签名算法
        Signature signature = Signature.getInstance (x509Certificate.getSigAlgName());
        // 获取私钥
        PrivateKey privateKey = getPrivateKeyByKeyStore(keyStorePath, alias, password);
        // 初始化签名，由私钥构建
        signature.initSign(privateKey);
        signature.update(sign);
        return signature.sign();
    }
    /**
     * 验证签名
     * @param data 数据
     * @param sign 签名
     * @param certificatePath 证书路径
     * @return boolean 验证通过为真
     * @throws Exception
```

更多资源请访问稀酷客(www.ckook.com)

```
     */
    public static boolean verify(byte[] data, byte[] sign, String
    certificatePath) throws Exception {
            // 获得证书
            X509Certificate x509Certificate = (X509Certificate) getCertificate(certificatePath);
            // 由证书构建签名
            Signature signature = Signature.getInstance(x509Certificate.getSigAlgName());
            // 由证书初始化签名，实际上是使用了证书中的公钥
            signature.initVerify(x509Certificate);
            signature.update(data);
            return signature.verify(sign);
    }
}
```

我们假定密钥库文件zlex.keystore存储在D盘根目录，数字证书文件zlex.cer也存储在D盘根目录。对上述代码做验证，如代码清单10-31所示。

代码清单10-31 基于密钥库和数字证书的加密/解密和签名/验证操作测试用例

```
import static org.junit.Assert.*;
import org.apache.commons.codec.binary.Hex;
import org.junit.Test;
/**
 * 证书校验
 * @author 梁栋
 * @version 1.0
 */
public class CertificateCoderTest {
    private String password = "123456";
    private String alias = "www.zlex.org";
    private String certificatePath = "d:/zlex.cer";
    private String keyStorePath = "d:/zlex.keystore";
    /**
     * 公钥加密——私钥解密
     * @throws Exception
     */
    @Test
    public void test1() throws Exception {
            System.err.println("公钥加密——私钥解密");
            String inputStr = "数字证书";
            byte[] data = inputStr.getBytes();
            // 公钥加密
            byte[] encrypt = CertificateCoder.encryptByPublicKey(data, certificatePath);
            // 私钥解密
            byte[] decrypt = CertificateCoder.decryptByPrivateKey(encrypt,
            keyStorePath, alias, password);
            String outputStr = new String(decrypt);
            System.err.println("加密前:\n" + inputStr);
```

更多资源请访问稀酷客(www.ckook.com)

```
        System.err.println("解密后:\n" + outputStr);
        // 验证数据一致
        assertArrayEquals(data, decrypt);
    }
    /**
     * 私钥加密——公钥解密
     * @throws Exception
     */
    @Test
    public void test2() throws Exception {
        System.err.println("私钥加密——公钥解密");
        String inputStr = "数字签名";
        byte[] data = inputStr.getBytes();
        // 私钥加密
        byte[] encodedData = CertificateCoder.encryptByPrivateKey(data,
        keyStorePath, alias, password);
        // 公钥加密
        byte[] decodedData = CertificateCoder.decryptByPublicKey(encodedData,
        certificatePath);
        String outputStr = new String(decodedData);
        System.err.println("加密前:\n" + inputStr);
        System.err.println("解密后:\n" + outputStr);
        assertEquals(inputStr, outputStr);
    }
    /**
     * 签名验证
     * @throws Exception
     */
    @Test
    public void testSign() throws Exception {
        String inputStr = "签名";
        byte[] data = inputStr.getBytes();
        System.err.println("私钥签名——公钥验证");
        // 产生签名
        byte[] sign = CertificateCoder.sign(data, keyStorePath, alias, password);
        System.err.println("签名:\n" + Hex.encodeHexString(sign));
        // 验证签名
        boolean status = CertificateCoder.verify(data, sign, certificatePath);
        System.err.println("状态:\n" + status);
        // 校验
        assertTrue(status);
    }
}
```

观察控制台输出，“公钥加密，私钥解密”输出如下所示：

```
公钥加密——私钥解密
加密前:
数字证书
```

更多资源请访问稀酷客(www.ckook.com)

```
解密后：
数字证书
```

“私钥加密，公钥解密”输出如下所示：

```
私钥加密--公钥解密
加密前：
数字签名
解密后：
数字签名
```

双向加密/解密操作完全通过！

我们再来关注签名/验证操作，控制台输出如下所示：

```
私钥签名--公钥验证
签名：
0f0344c3db3d3349a9037ce98468ae8ea1aa9fd70f47139928c2e986185401971e0382d991b56893
e07190304079955c79a06e1750045e27723e29d538664bd92c6dddeb0a6b51a76c32de5b5bdb8d2be493
dfbabc8d94855b215d0b5b775b6008f44caa0c4f6ca574cb5cea8c69427bbce07c662fcdcaef8a22e2dd
50d462cb
状态：
true
```

我们在无需知晓密钥和算法的前提下，完成了数据加密/解密和签名/验证操作。

10.5 应用举例

随着网络应用的普及，数字证书主要应用于各种电子商务活动和电子政务活动。电子商务活动主要包括网上缴费、网上炒股、网上购物等；电子政务活动主要包括网上招标投标、网上签约、网上公文传送和网上报关等。

根据相应的活动需要数字证书也分为多种类型，主要包括：个人数字证书、单位数字证书、单位员工数字证书、服务器数字证书、VPN数字证书、WAP数字证书、代码签名数字证书和表单签名数字证书等。

许多软件公司在自己的软件中嵌入数字证书，可用于验证签名，从而有效地遏制软件的盗版。同时，可以使用数字证书实现加密/解密操作，保护私密数据，进行加密网络交互。

数字证书常常与传输层SSL/TLS协议共同构建应用层HTTPS协议，确保网络交互安全。目前，各大电子商务网站均使用HTTPS协议，确保网络交易安全。各种Web Service为确保数据交互的安全性，通常使用HTTPS协议加强系统安全性。

10.6 小结

数字证书是密码学领域的终极武器，它集合了多种密码学算法：自身带有公钥信息，可完成相应的加密/解密操作，同时，还带有数字签名，可鉴别消息来源；且自身带有消息摘要信息，

更多资源请访问稀酷客(www.ckook.com)

可验证证书的完整性；由于证书本身含有用户身份信息，因而具有认证性。OSI参考模型五类安全服务除访问控制服务外，均可通过数字证书实现。

数字证书要经权威数字证书颁发认证机构签发，VeriSign（http://www.verisign.com/）、GeoTrust（http://www.geotrust.com/）和Thawte（http://www.thawte.com/）是国际权威数字证书颁发认证机构的“三巨头”。其中，应用最为广泛的是VeriSign签发的电子商务用数字证书。由“三巨头”颁发的数字证书需要向用户收取昂贵的申请和维护费用，而CAcert（http://www.cacert.org/）则是一个为我们提供免费数字证书颁发的国际组织。

KeyTool和OpenSSL 是我们常用的数字证书管理工具。KeyTool是Java中的数字证书管理工具，用于数字证书的申请、导入、导出和撤销等操作。OpenSSL（http://www.openssl.org/）是一个开放源代码软件包，由Eric A. Young 和 Tim J. Hudson等人编写，实现了SSL及相关加密技术，是最常用的证书管理工具。

OpenSSL功能远胜于KeyTool，可用于根证书、服务器证书和客户证书的管理，常用来构建双向认证服务。

Java 6完全支持X.509标准的数字证书，可配合使用密钥库和数字证书构建加密交互网络平台。

更多资源请访问稀酷客(www.ckook.com)

第11章

终极装备——安全协议

网银平台通常少不了数字证书，更少不了安全协议——HTTPS协议。HTTPS协议实际上基于SSL/TLS的HTTP协议，位于应用层，简单地说，HTTPS=HTTP+SSL/TLS。SSL/TLS协议本身是带有加密信息的传输层协议，数字证书正是为这种协议提供相关加密/解密信息。

11.1 安全协议简述

HTTPS协议和SSL/TLS协议分属TCP/IP参考模型中的应用层和传输层。简单地说，HTTPS就是附加了SSL/TLS协议的HTTP协议。HTTPS协议为数字证书提供了最佳的应用环境！

11.1.1 HTTPS协议

HTTPS（Hypertext Transfer Protocol over Secure Socket Layer）协议是Web上最为常用的安全访问协议。简单地说，HTTPS就是HTTP安全版，HTTPS是基于SSL/TLS的HTTP协议，或者说HTTPS= SSL/TLS+HTTP。

相比于SSL/TLS协议，HTTPS协议我们更为熟悉。在生活中，我们常常需要访问基于HTTPS协议的Web网站，如图11-1所示。



图11-1 招行信用卡网银页面

图11-1是招行“个人银行大众版”页面，这里使用的不是HTTP协议，而是HTTPS协议。在地址栏中，还有一个醒目的小锁图标，表明这是一个安全网站。单击小锁图标，获得网站标识信息，如图11-2所示。



图11-2 网站标识信息

由图11-2得知，该网站使用的证书由VeriSign（https://www.verisign.com/）签发，单击“查看证书”，获得证书相关信息如图11-3所示。

招行数字证书由VeriSign签发，颁发给域名为“pbsz.ebank.cmbchina.com”的招行网银，这个域名正是招行信用卡网银域名。点击颁发者说明，可见VeriSign官网信息如图11-4所示。

更多资源请访问稀酷客(www.ckook.com)

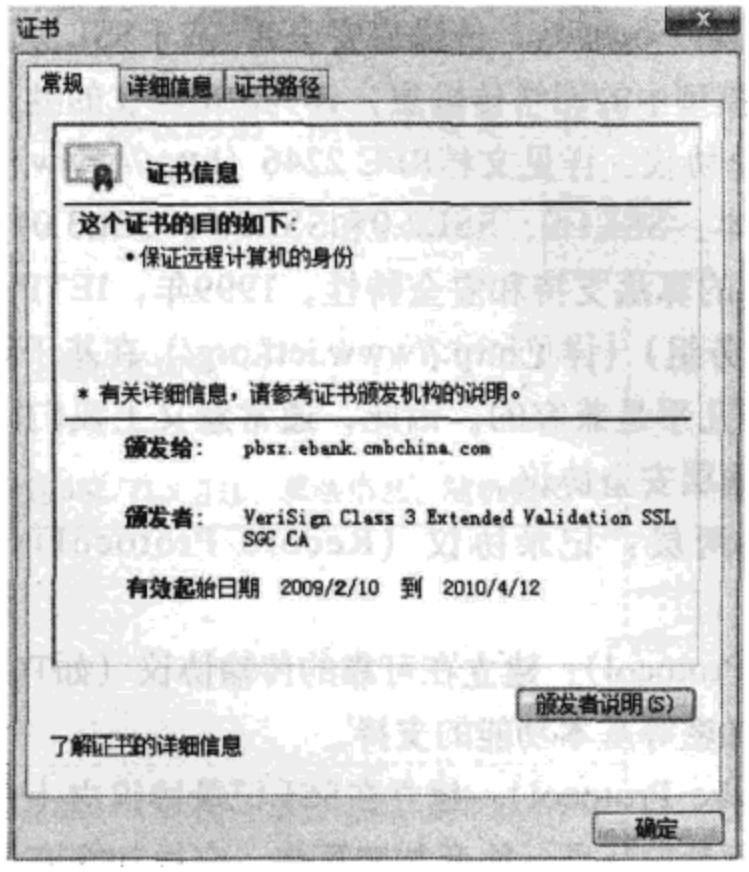


图11-3 招行数字证书

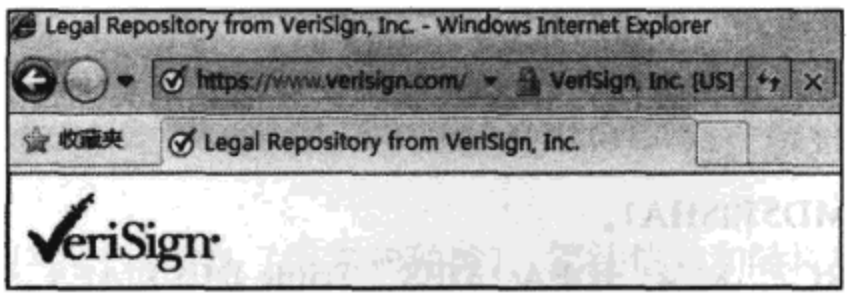


图11-4 VeriSign官网

目前，VeriSign签发的数字证书广泛应用于电子商务平台。

HTTPS协议常常在服务器中配置，如HTTP服务器Apache（http://httpd.apache.org/）和JSP服务器Tomcat（http://tomcat.apache.org/），通过配置SSL/TLS协议构建基于HTTPS协议的服务器。在Tomcat服务器中配置HTTPS协议如图11-5所示。

稍后，我们将构建一个简单基于HTTPS的Web应用！

```
<Connector
    SSLEnabled="true"
    URIEncoding="UTF-8"
    clientAuth="false"
    keystoreFile="conf/zlex.keystore"
    keystorePass="123456"
    truststoreFile="conf/zlex.keystore"
    truststorePass="123456"
    maxThreads="150"
    port="443"
    protocol="HTTP/1.1"
    scheme="https"
    secure="true"
    sslProtocol="TLS" />
```

图11-5 在Tomcat服务配置HTTPS协议

11.1.2 SSL/TLS协议

SSL/TLS协议包含两个协议：SSL（Secure Socket Layer，安全套接字层）和TLS（Transport Layer Security，传输层安全）协议。

- SSL（Secure Socket Layer，安全套接字层）：由Netscape（网景）公司研发，位于TCP/IP参考模型中的网络传输层，作为网络通讯提供安全及数据完整性的一种安全协议。

更多资源请访问稀酷客(www.ckook.com)

- TLS（Transport Layer Security，传输层安全）：基于SSL协议之上的通用化协议，它同样位于TCP/IP参考模型中的网络传输层，作为SSL协议的继任者，成为下一代网络安全性和数据完整性安全协议。详见文档RFC 2246（http://www.ietf.org/rfc/rfc2246.txt）。

目前，SSL共有3个版本：SSL1.0、SSL2.0和SSL3.0。SSL3.0规范在1996年3月正式发布，较之前2个版本提供了更多的算法支持和安全特性。1999年，IETF（The Internet Engineering Task Force，互联网工程任务组）（详见http://www.ietf.org/）在基于SSL3.0协议的基础上发布了TLS1.0，TLS1.0与SSL3.0几乎是兼容的。因此，通常意义上我们提到的SSL/TLS协议指的是SSL3.0或TLS1.0的网络传输层安全协议。

SSL/TLS协议可分为两层：记录协议（Record Protocol）和握手协议（Handshake Protocol）。

- 记录协议（Record Protocol）：建立在可靠的传输协议（如TCP）之上，为高层协议提供数据封装、压缩、加密等基本功能的支持。
- 握手协议（Handshake Protocol）：建立在SSL记录协议之上，用于在实际的数据传输开始前，通讯双方进行身份认证、协商加密算法、交换加密密钥等。

握手协议较为底层，不好理解，好在目前主流计算机语言（如Java语言）的开发者已经将这些协议的处理封装得透明，无需我们关心。

SSL/TLS协议涉及多种加密算法，包含消息摘要算法、对称加密算法、非对称加密算法，以及数字签名算法。

- 消息摘要算法：MD5和SHA1。
- 对称加密算法：RC2、RC4、IDEA、DES、Triple DES和AES。
- 非对称加密算法：RSA和Diffie-Hellman（DH）。
- 数字签名算法：RSA和DSA。

SSL/TLS协议利用密码学算法在互联网上提供端点身份认证和通信保密，完全基于PKI，有较高的安全性。因此，SSL/TLS协议成了网络上最常用的网络传输层安全保密通讯协议，众多电子邮件、网银、网上传真都通过SSL/TLS协议确保数据传输安全。随着卫星和无线网络的发展，WAP安全逐渐得到重视。受限于手机及手持设备的处理和存储能力，WAP论坛（http://www.wapforum.org/）在TLS的基础上进行了简化，制定了WTLS（Wireless Transport Layer Security，无线传输层安全）协议。

11.2 模型分析

经过SSL/TLS握手协议交互后，数据交互双方确定了本次会话使用的对称加密算法以及密钥，由此开始基于对称加密算法的加密数据交互。

握手协议较为复杂，我们将分析主要环节，阐述服务器端和客户端构建加密交互的相关流程。

更多资源请访问稀酷客(www.ckook.com)

11.2.1 协商算法

服务器端和客户端进行握手协议的第一阶段主要是协商算法，如图11-6所示。

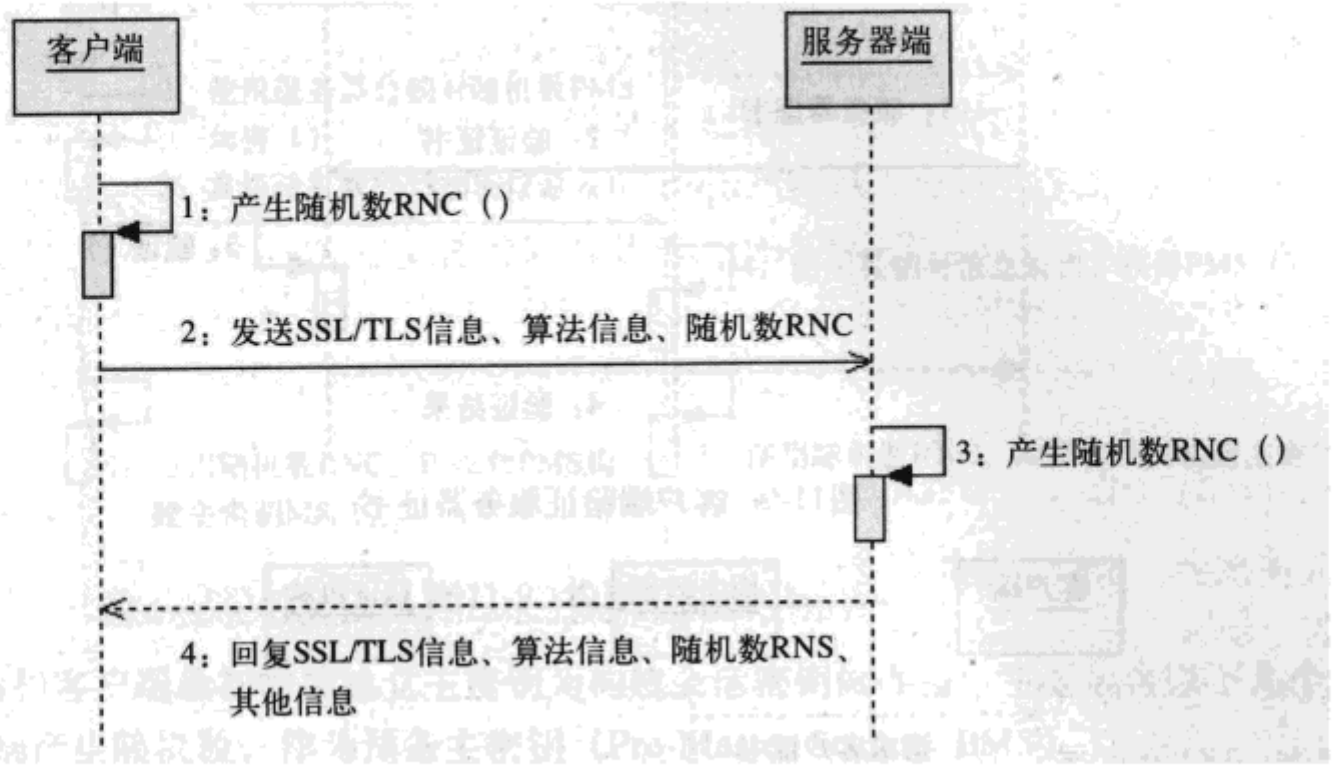


图11-6 对等协商加密算法

服务器端和客户端在进行握手协议第一阶段时主要是商榷加密算法，主要包含以下几个步骤：

1）客户端产生随机数RNC（Random Number Client），这个随机数将为后续构建密钥做准备。

2）客户端将自身支持的SSL信息（版本和种类）、算法信息和随机数RNC发送到服务器端。

3）服务器端得到客户端请求后，产生相应的随机数RNS（Random Number Server），这个随机数为后续构建密钥做准备。

4）服务器端将自身支持的SSL信息（版本和种类）、算法信息、随机数RNS和其他信息回应到客户端。其他信息包括服务器证书，甚至包含获取客户端证书的请求。

这时，服务器端和客户端已经确认两方交互时所使用的加密算法。

11.2.2 验证证书

如果服务器端回复客户端时带有其他信息，则进入数字证书验证阶段，如图11-7和图11-8所示。

服务器端下发服务器证书给客户端后，由客户端验证该证书，主要包含以下几个步骤：

1）服务器回复客户端响应时带有服务器证书。

2）客户端将该证书发送至认证机构。

3）认证机构鉴别该证书。

4）认证机构回应客户端验证结果，如果验证失败将同时得到警告信息。

这时，服务器端身份得以认证，客户端和服务器端可以进行以服务器端单向认证为基础的加密交互。如果服务器端对于客户端身份有要求，下发服务器证书的同时要求客户端提供证书，

更多资源请访问稀酷客(www.ckook.com)

如图11-8所示。

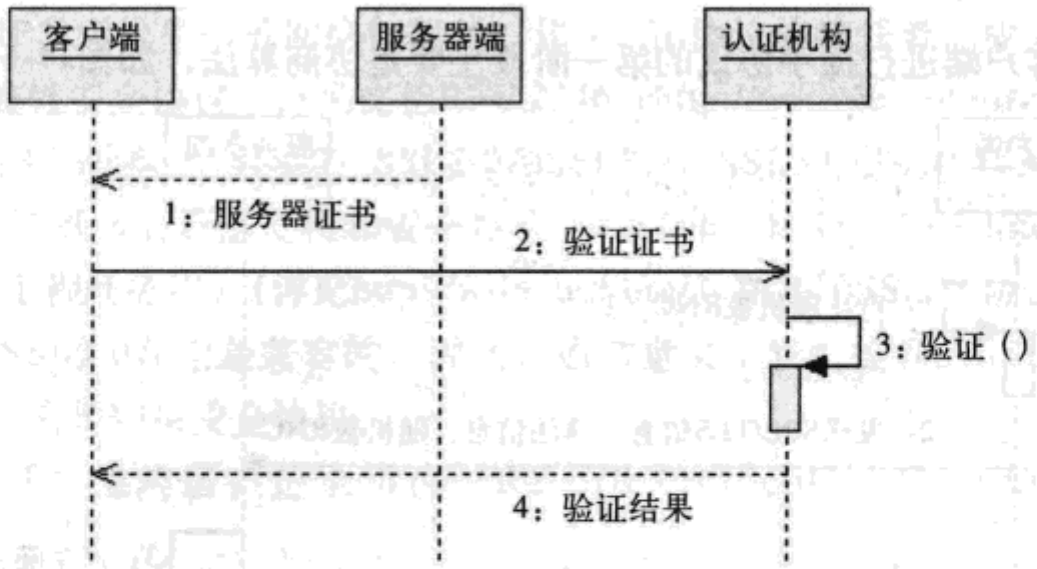


图11-7 客户端验证服务器证书

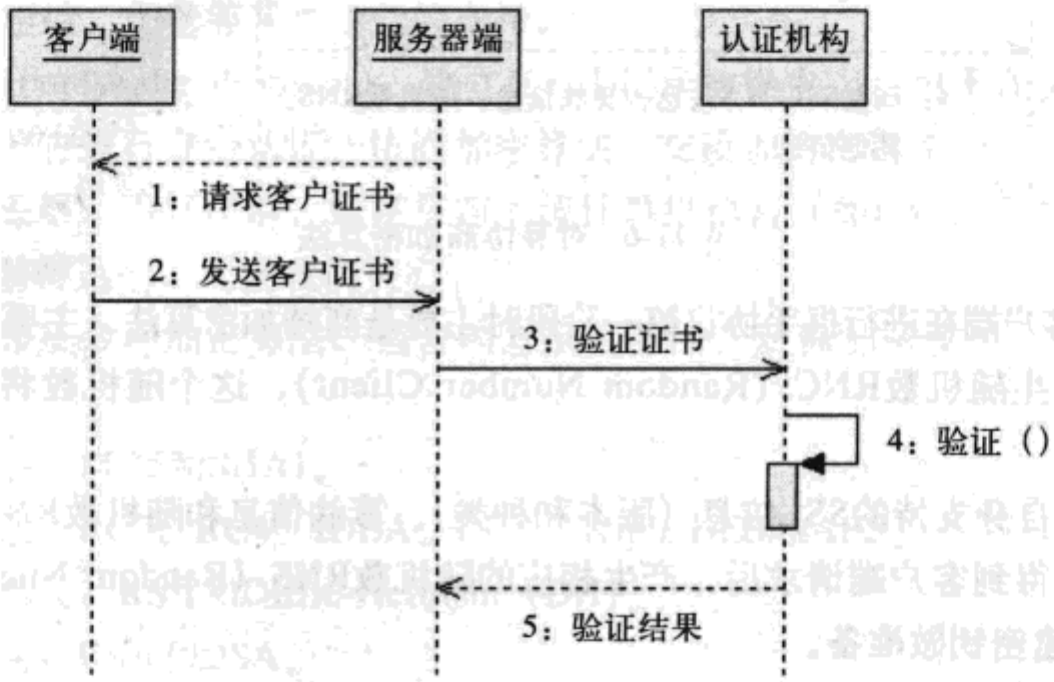


图11-8 服务器端验证客户证书

服务器端要求客户端提供客户证书，将构建基于客户端和服务器端两方的双向认证基础。服务器端验证客户证书，主要包含以下几个步骤：

1）服务器端请求客户证书。

2）客户端发送客户证书。

3）服务器将客户证书发送至认证机构验证。

4）认证机构验证证书。

5）认证机构返回验证结果。

通常，客户证书认证不一定必需。如果客户端和服务器端两方都可以确认，就可进行以双向认证为基础的加密交互。双向认证是电子商务确保安全的必要环节。

11.2.3 产生密钥

当服务器端和客户端经过上述一系列操作后，开始密钥构建交互，如图11-9和图11-10所示。

更多资源请访问稀酷客(www.ckook.com)

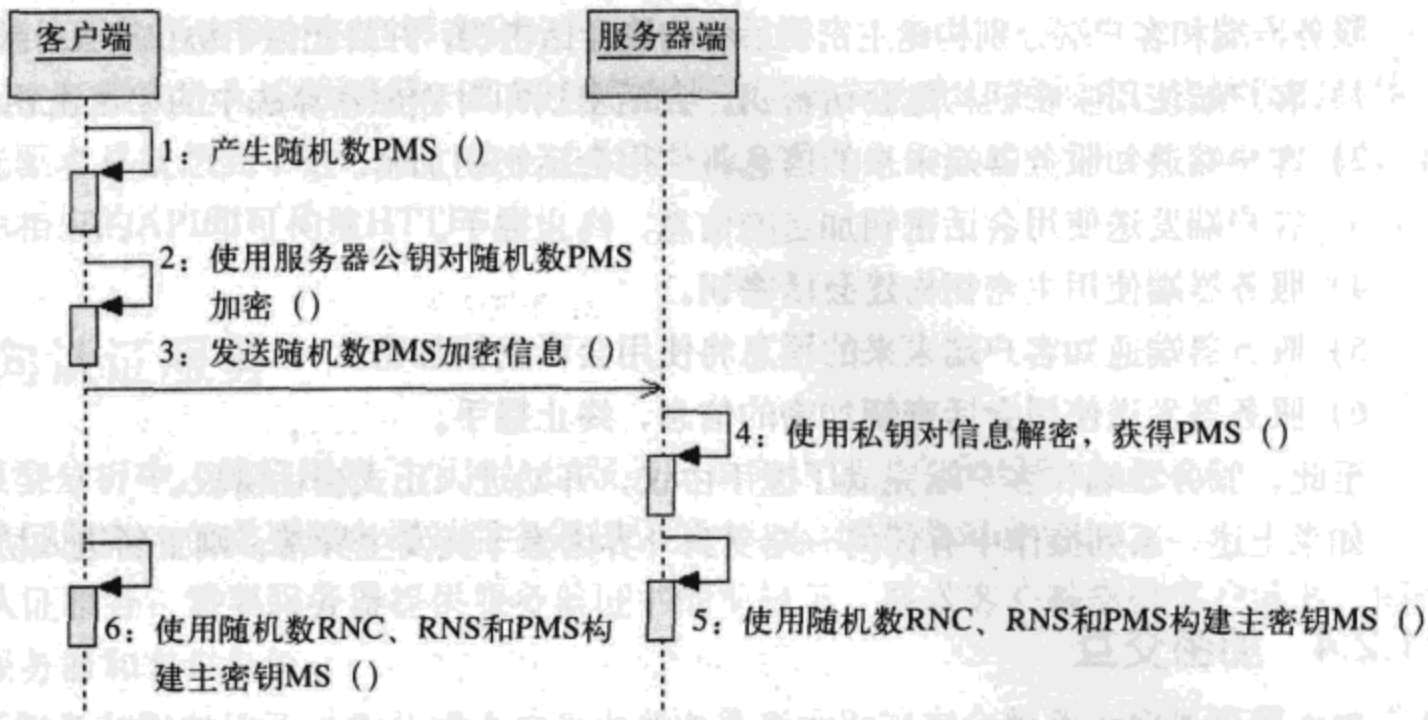


图11-9 构建主密钥MS

服务器端和客户端最初需要建立主密钥为构建会话密钥做准备，主要包含以下几个步骤：

1）客户端产生随机数，作为预备主密钥（Pre-Master Secret，PMS）。

2）客户端使用服务器证书中的公钥对随机数PMS加密。

3）客户端将PMS加密信息发送到服务器端。

4）服务器使用私钥对信息解密获得PMS信息。

5）客户端使用随机数RNC、RNS和PMS构建主密钥（Master Secret，MS）。

6）服务器端使用随机数RNC、RNS和PMS构建主密钥MS。

上述步骤5、6不存在次序关系，实际操作中异步完成。

完成主密钥构建操作后，服务器端和客户端将建立会话密钥，即将完成握手协议，如图11-10所示。

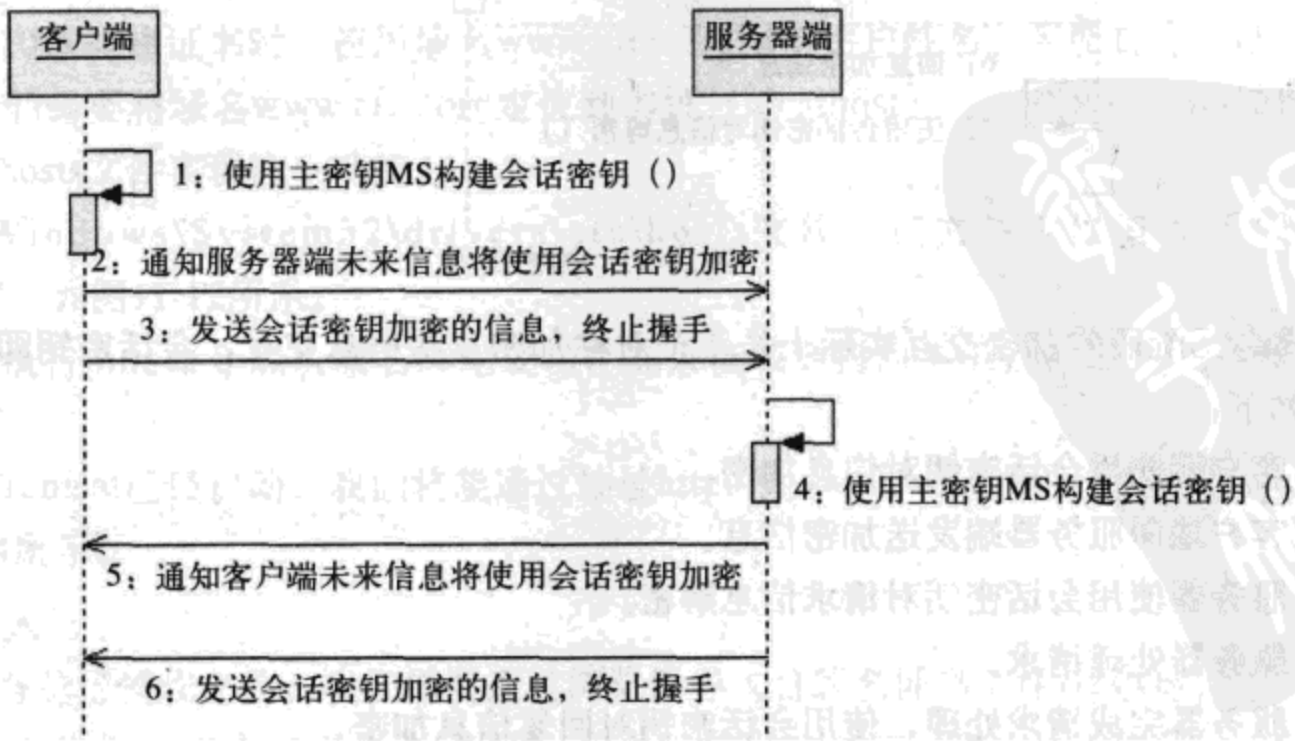


图11-10 构建会话密钥

更多资源请访问稀酷客(www.ckook.com)

服务器端和客户端分别构建主密钥后将构建会话密钥，并终止握手协议交互，主要步骤包括：

1）客户端使用主密钥构建会话密钥。会话密钥即对称加密算法中的秘密密钥。

2）客户端通知服务器端未来的信息将使用会话密钥加密。

3）客户端发送使用会话密钥加密的信息，终止握手。

4）服务器端使用主密钥构建会话密钥。

5）服务器端通知客户端未来的信息将使用会话密钥加密。

6）服务器发送使用会话密钥加密的信息，终止握手。

至此，服务器端和客户端完成了握手协议，开始进入正式会话阶段。

如果上述一系列操作中有任何一端受到外界因素干扰发生异常，则重新进入协商算法阶段。

11.2.4 加密交互

进入正式会话阶段后，服务器端和客户端将使用会话密钥进行加密交互，如图11-11所示。

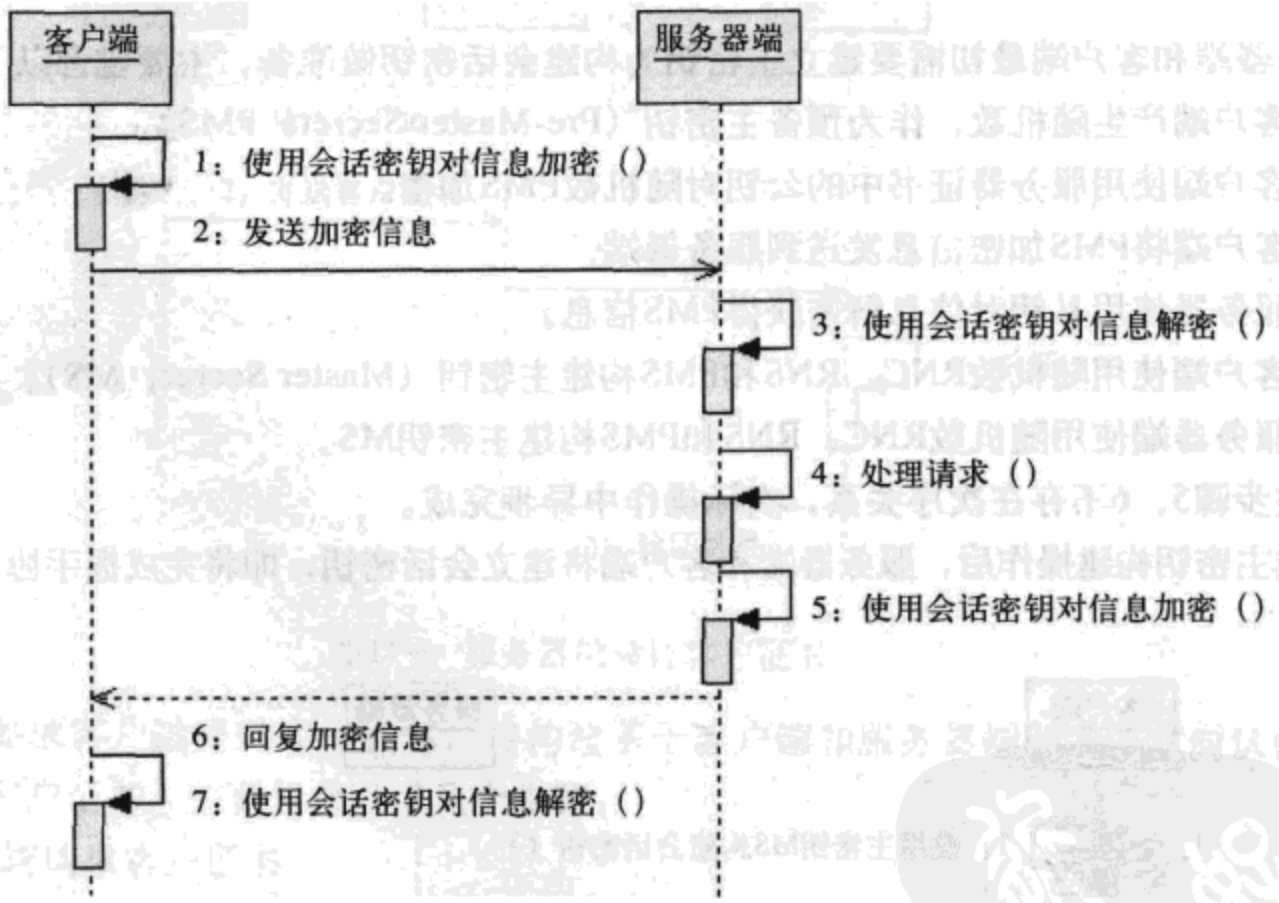


图11-11 加密交互

正式会话阶段的加密交互实际上是基于对称加密算法信息交互，会话密钥即秘密密钥，主要步骤如下：

1）客户端使用会话密钥对信息加密。

2）客户端向服务器端发送加密信息。

3）服务器使用会话密钥对请求信息解密。

4）服务器处理请求。

5）服务器完成请求处理，使用会话密钥对回复信息加密。

6）服务器回复加密信息。

更多资源请访问稀酷客(www.ckook.com)

7）客户端使用会话密钥对信息解密。

握手协议交互着实让人难以理解，理解上述协议交互尚有难度，更别说这些协议的具体实现。智慧的先驱者早就想到了这一点，将上述实现封装在SSL（Security Socket Layer）层，我们只需要调用相应的API即可构建HTTPS协议。

11.3 单向认证服务

在上述模型分析中，我们提到了两种认证服务：单向认证服务和双向认证服务。

- 单向认证服务：仅需要服务器端服务器提供证书，验证服务器身份。
- 双向认证服务：需要服务器提供服务器证书的前提下，要求客户端提供客户证书，同时验证服务器和客户身份。

单向认证服务和双向认证服务是网络交互平台中最高级别的安全服务，广泛应用于电子商务等领域。

本文以配置Tomcat安全服务为例，介绍如何搭建基于单向认证服务和双向认证服务的网络交互平台。读者朋友可以从Apache官网（http://tomcat.apache.org/）下载Tomcat当前的最新版本。

为便于演示，本文使用KeyTool工具构建自签名证书，搭建单向认证服务。实际应用时，请读者朋友使用经由CA机构签发的数字证书。

本文将使用密钥库文件zlex.keystore和数字证书文件zlex.cer，相关实现请参考第10章内容。

11.3.1 准备工作

单向认证服务实现不需要实现任何代码，我们仅需要对Tomcat做细微调整，完成HTTPS协议配置和密钥库配置，即可完成服务构建工作。

1. 域名绑定

我们在第10章构建证书时，使用域名www.zlex.org作为用户姓名。为使自签名证书通过浏览器验证，我们需要将域名www.zlex.org定位到本机（localhost）上。在Windows操作系统中可以通过修改hosts文件实现这一功能。

打开C:\Windows\System32\drivers\etc\hosts文件，在文件末尾追加“127.0.0.1 www.zlex.org”，如图11-12所示。

我们可以执行ping命令确认域名绑定成功，如果该域名指向本机，就说明域名绑定成功，如图11-13所示。

如果这时Tomcat已经启动，我们将能通过地址http://www.zlex.org:8080/访问Tomcat欢迎页面，如图11-14所示。

2. 证书导入

在进行服务验证操作之前，我们需要通过浏览器导入自签名证书文件zlex.cer。

以IE浏览器为例，点击“工具”菜单，在弹出菜单中点击“Internet选项”，在弹出的“Internet选项”对话框中选择“内容”选项卡。如图11-15所示。

更多资源请访问稀酷客(www.ckook.com)

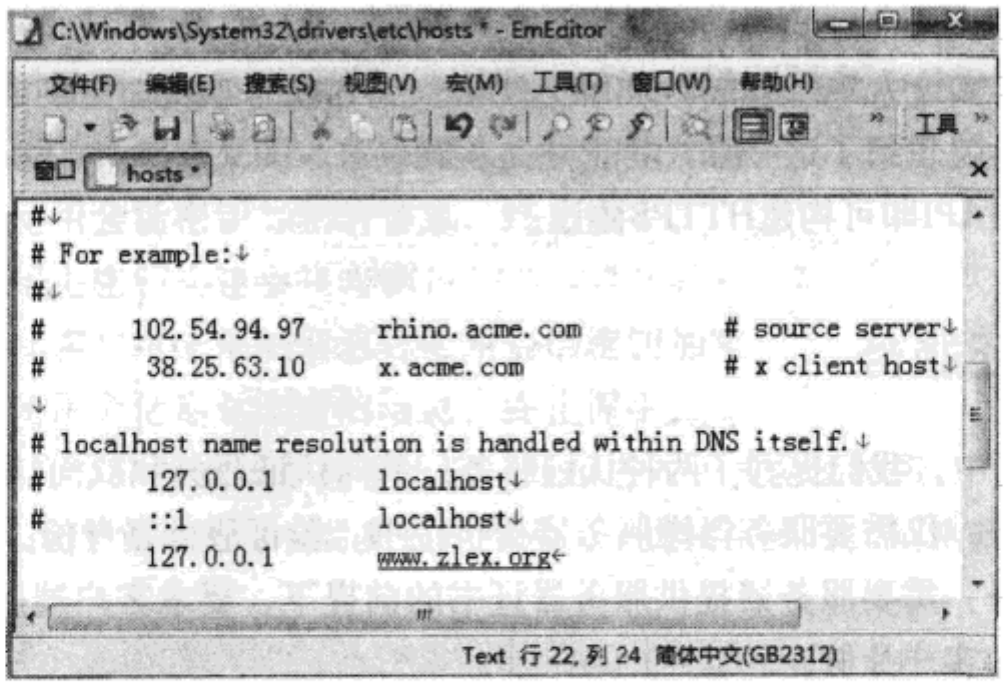


图11-12 配置hosts文件

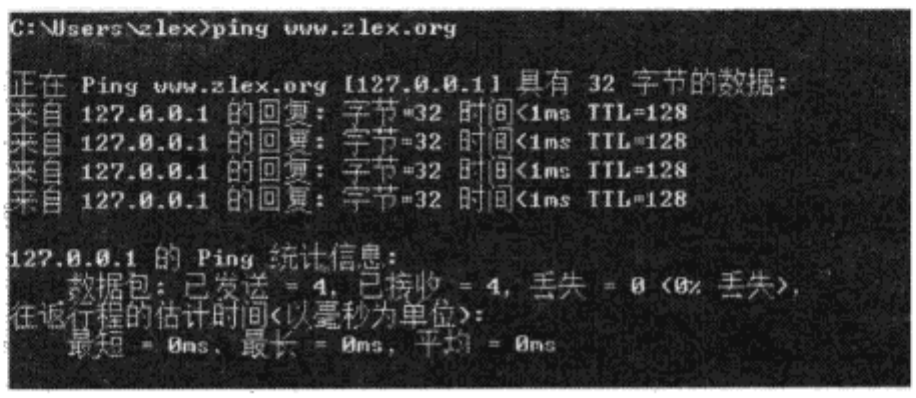


图11-13 验证域名绑定

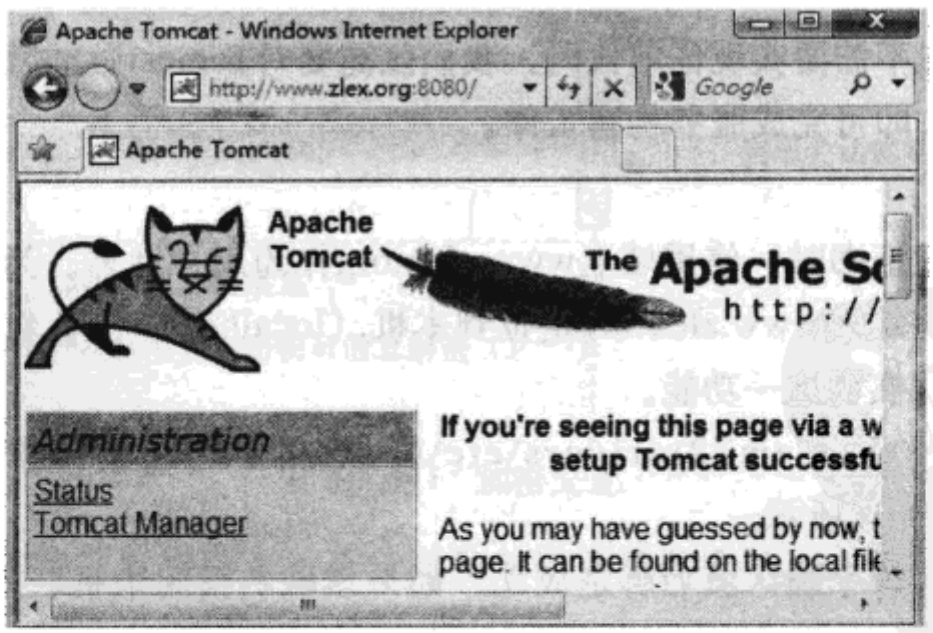


图11-14 验证域名绑定验证

在“内容”选项卡证书栏中点击“证书”按钮。在弹出的“证书”对话框中选择“受信任的根证书颁发机构”选项卡，点击“导入”按钮。

在弹出的“证书导入向导”对话框中，点击“下一步”，单击“浏览”按钮选择自签名证书文件——zlex.cer文件，如图11-16所示。

更多资源请访问稀酷客(www.ckook.com)

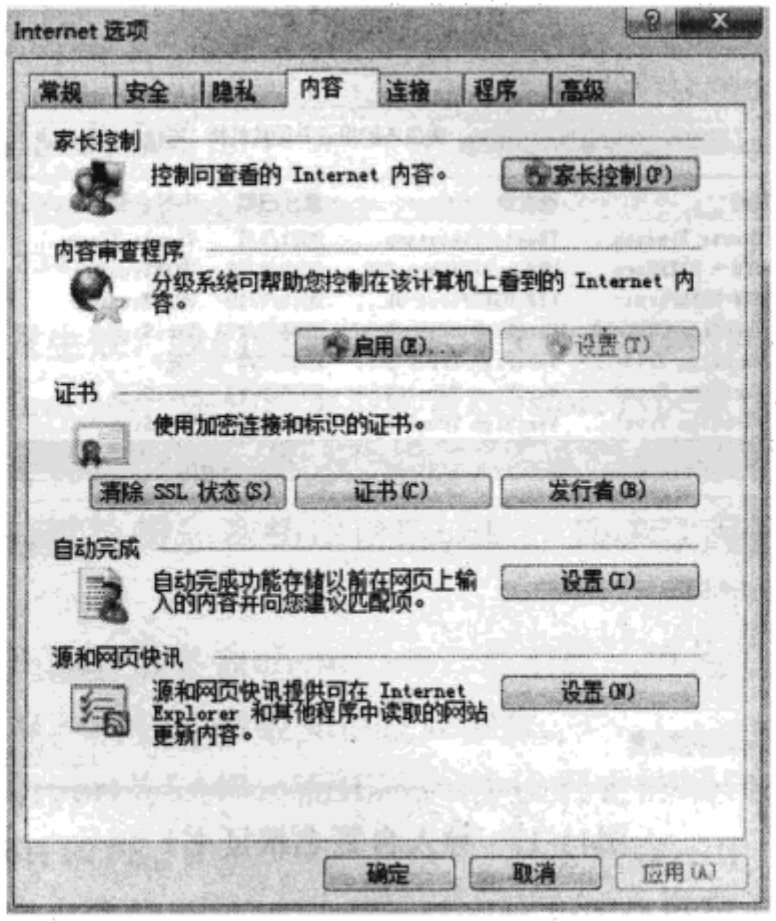


图11-15 Internet选项-内容选项卡

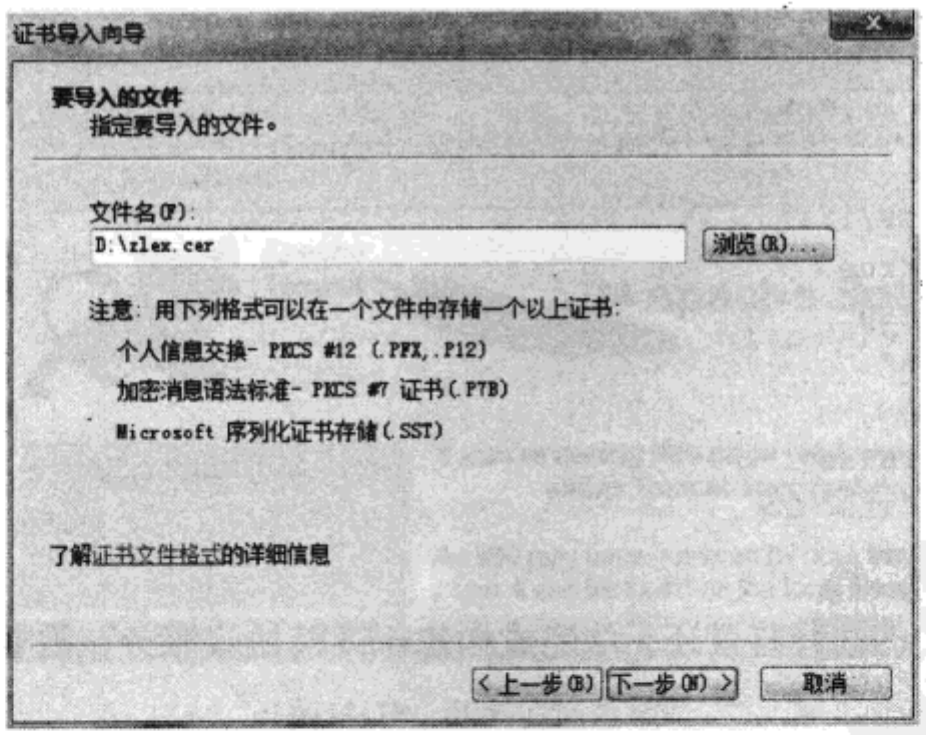


图11-16 证书导入向导

继续点击“下一步”按钮，直至完成证书导入。最终，我们将在“受信任的根证书颁发机构”选项卡中找到我们导入的自签名证书，如图11-17所示。

如果我们此处使用的是经过国际CA机构签发的数字证书，将无需导入。

3. 服务器配置

相信大多数读者朋友对于Tomcat都比较了解，配置SSL/TLS协议需要修改server.xml文件。server.xml文件位于Tomcat的conf目录中，属于Tomcat核心配置文件。相关配置可参见官网文档(http://tomcat.apache.org/tomcat-6.0-doc/ssl-howto.html)。

更多资源请访问稀酷客(www.ckook.com)

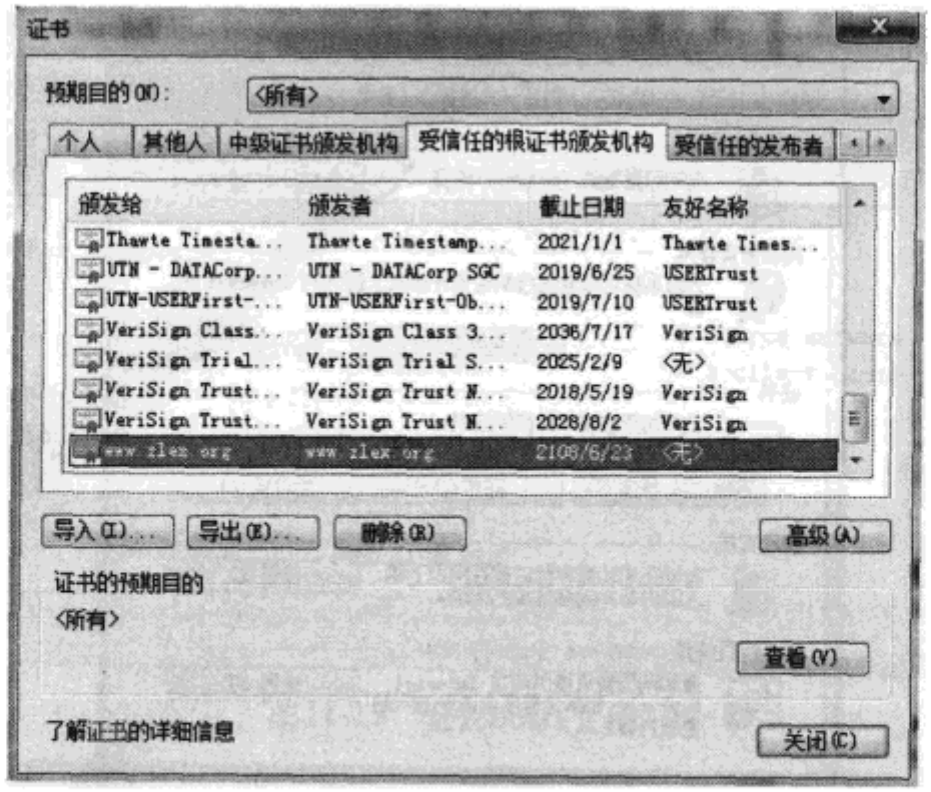


图11-17 导入自签名根证书1

打开server.xml文件，并找到包含关键字“TLS”的一行，内容如代码清单11-1所示。

代码清单11-1 SSL/TLS原始配置

```
<!--
<Connector
    port="8443"
    protocol="HTTP/1.1"
    SSLEnabled="true"
    maxThreads="150"
    scheme="https"
    secure="true"
    clientAuth="false"
    sslProtocol="TLS" />
-->
```

仅仅将上述代码中的注释打开还无法完成构建HTTPS协议服务，需要配置相应的证书和密钥库等。

在第10章中，我们将签发的数字证书文件zlex.cer导入密钥库zlex.keystore中。现在我们需要将密钥库文件zlex.keystore放置在Tomcat的conf目录下，并将其配置在上述配置文件中，如代码清单11-2所示。

代码清单11-2 配置SSL/TLS单向认证

```
<Connector
    port="443"
    SSLEnabled="true"
    clientAuth="false"
    maxThreads="150"
```

更多资源请访问稀酷客(www.ckook.com)

```
protocol="HTTP/1.1"
scheme="https"
secure="true"
sslProtocol="TLS"
keystoreFile="conf/zlex.keystore"
keystorePass="123456" />
```

为使得HTTPS协议配置生效，我们需要将密钥库文件参数keystoreFile指向密钥库文件，并设定密钥库密码参数keystorePass，密钥库类型参数keystoreType默认值为“JKS”。

如果不显示配置信任库参数，信任库文件参数truststoreFile默认将指向密钥库文件，信任库密码参数truststorePass默认指向密钥库密码，信任库类型参数truststoreType默认值为“JKS”。

这里我们需要注意客户端验证参数clientAuth，当前默认值为“false”。构建双向认证服务时需将其置为“true”，并修改密钥库参数和信任库参数。

原有配置指定端口参数port为8443，而HTTPS协议绑定的端口是443，我们可以将其改为443，而无需通过端口访问HTTPS。

启动Tomcat，在浏览器中访问地址https://www.zlex.org/查看数字证书是否生效，如图11-18所示。

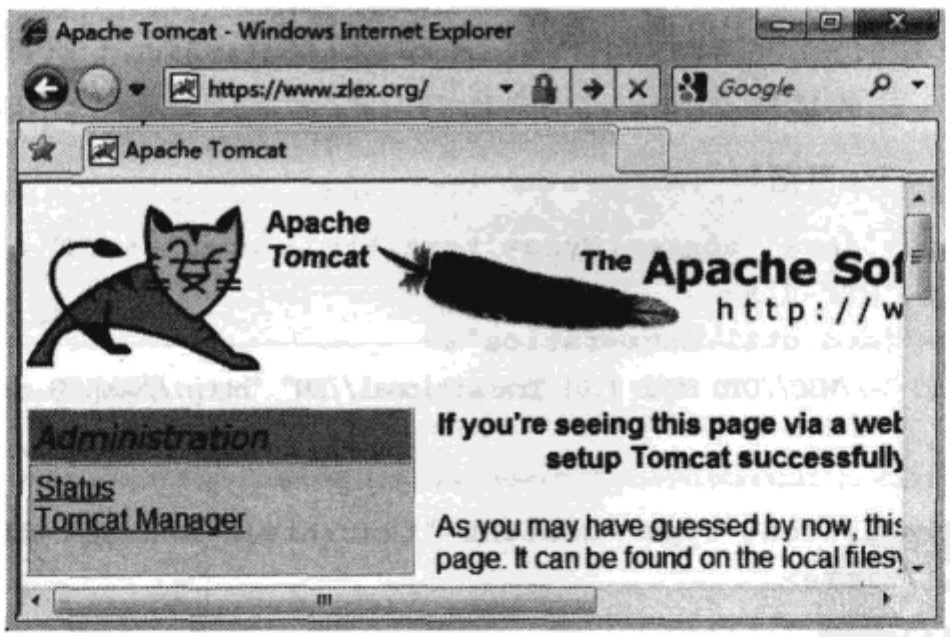


图11-18 证书验证生效

虽然这时我们连接的是本地服务器，但仍会感到联网有些迟钝（大约5秒钟）。这是因为浏览器访问CA机构验证该证书，并初始化加密/解密信息等，这一系列动作致使访问延时。

此时，如果未能正确导入自签名证书，将得到“证书错误”提示页面，如图11-19所示。

这是由于自签名证书的根证书没有通过CA验证而提示证书错误。我们可以点击“继续浏览此网站”转到下一页面，如图11-20所示。

此时，请导入根证书文件，并重新打卡IE浏览器访问上述地址。

更多资源请访问稀酷客(www.ckook.com)

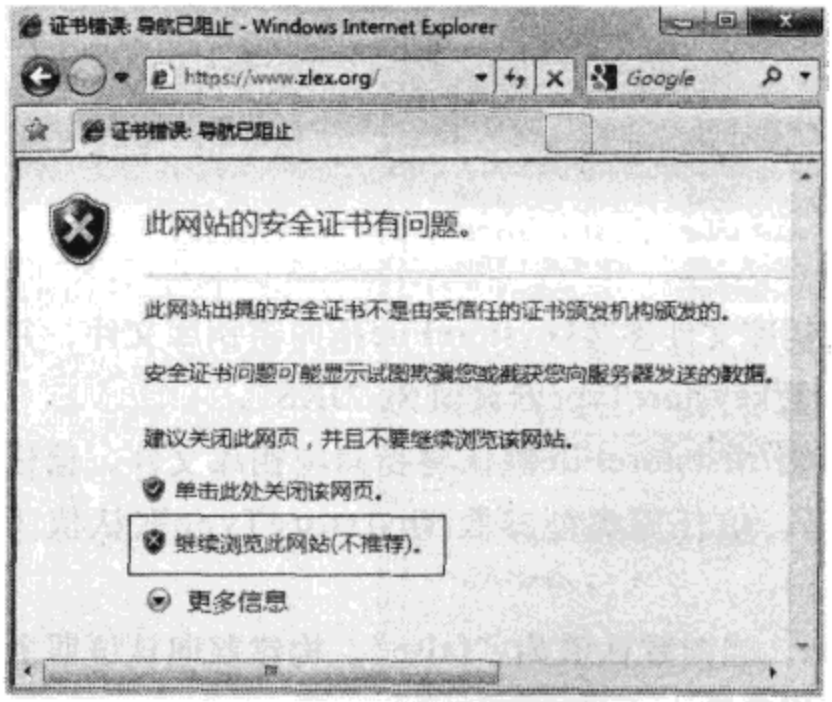


图11-19 证书错误提示1

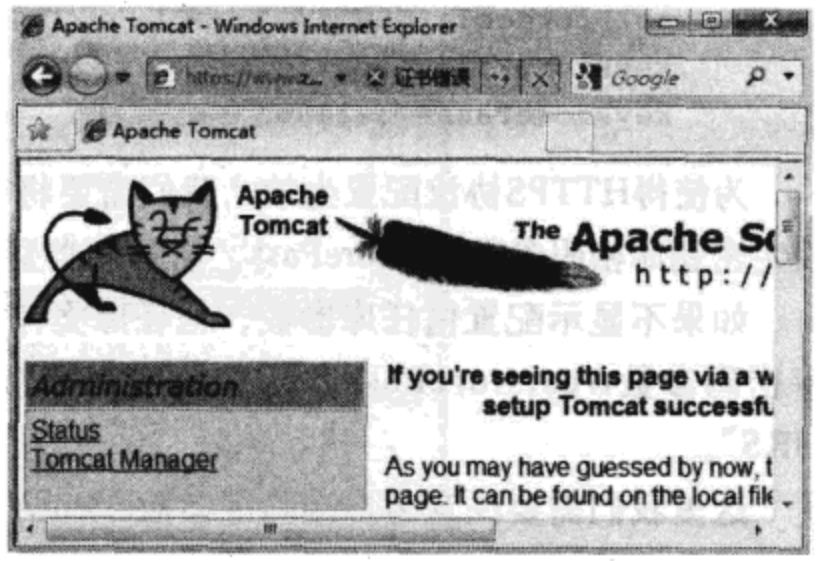


图11-20 证书错误提示2

11.3.2 服务验证

根据Tomcat官方文档提示，完成HTTPS协议架设后，可以从request的属性中获得有关加密的相关信息，如加密算法、密钥长度等。我们可以通过构建一个简单的应用来验证上述描述。

首先，我们需要构建一个JSP页面，遍历request的属性，并将其打印。完整实现如代码清单11-3所示。

代码清单11-3 index.jsp页面1

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ page import="java.util.Enumeration"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>zlex.org</title>
</head>
<body>
<p>request属性信息</p>
<pre>
<%
    for (Enumeration en = request.getAttributeNames(); en
                    .hasMoreElements();) {
            String name = (String) en.nextElement();
            out.println(name);
            out.println(" = " + request.getAttribute(name));
            out.println();
    }
%>
</pre>
```

更多资源请访问稀酷客(www.ckook.com)

```
</body>
</html>
```

接下来，我们需要创建一个最为普通的web.xml文件，如代码清单11-4所示。

代码清单11-4 web.xml文件

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    id="WebApp_ID"
    version="2.5">
    <display-name>ssl</display-name>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

最后，我们在Tomcat的webapps目录下创建一个ssl目录，并将上述index.jsp文件放在这个目录下。同时，在ssl目录下创建一个WEB-INF目录，并将web.xml文件放置其中。

现在，我们访问https://www.zlex.org/ssl/将得到request的全部属性值，如图11-21所示。

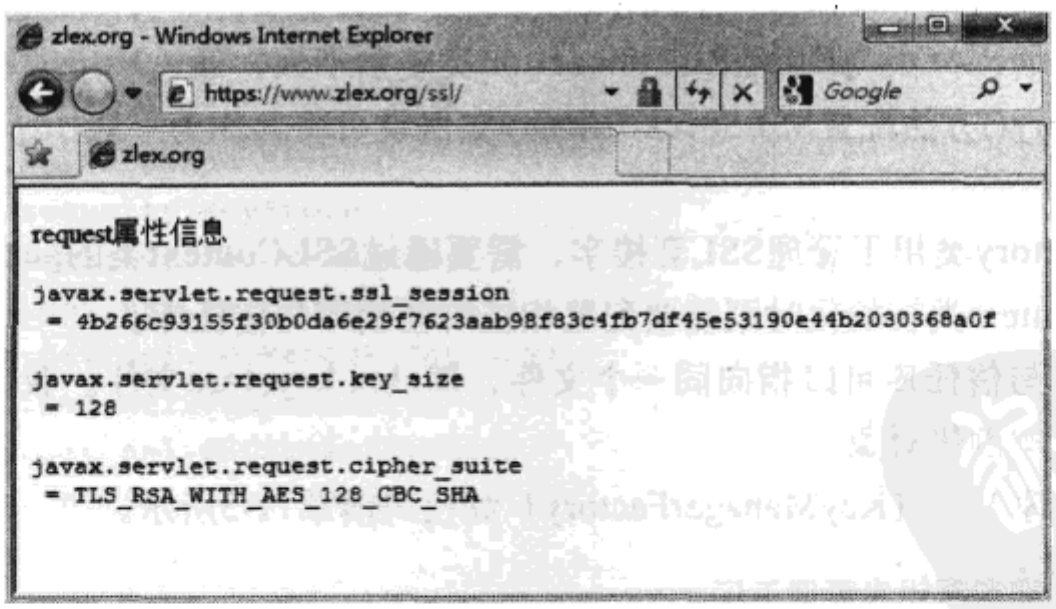


图11-21 SSL/TLS协议中的request属性1

在这里我们获得了3个有关SSL/TLS协议的request属性：

- javax.servlet.request.ssl_session：当前SSL/TLS协议的会话ID。
- javax.servlet.request.key_size：当前加密算法所使用的密钥长度。
- javax.servlet.request.cipher_suite：当前SSL/TLS协议所使用的加密套件。

通过javax.servlet.request.cipher_suite属性值（TLS_RSA_WITH_AES_128_CBC_SHA），我们获知本次连接所使用的协议为TLS协议，非对称加密算法为RSA算法，对称加密算法为AES

更多资源请访问稀酷客(www.ckook.com)

算法，消息摘要算法为SHA1。其中，对称加密算法要求密钥长度为128位，且工作模式为CBC模式。

对于使用何种协议、对称加密算法以及消息摘要算法，需由客户端与服务器端交互通过SSL/TLS握手协议确定。不同的客户端同一时间访问同一服务时，将有可能使用不同的协议或算法，唯一可以确定的是通过数字证书确定的非对称加密算法。

这里要说明一点，使用非对称加密算对数据进行加密/操作的效率相当低，而使用对称加密算法进行加密/操作的效率相当高。合理的解决办法是使用非对称加密算法传递对称加密算法的密钥，使用对称加密算法对数据加密。

11.3.3 代码验证

如果我们需要通过一个自己实现的客户端访问上述服务，该如何去做呢？我们是否需要了解SSL/TLS协议？我们是否需要完成相应的加密/解密操作，并对数据做验证/签名呢？

对于上述问题，Java语言的开发者已经将其封装。我们只需要获得数字证书，并将其导入密钥库应用相应的Java API，即可进行加密交互。

读者朋友需要先阅读第3章相关内容，了解SSL/TLS协议相关Java API。

这里我们直接使用本文中的密钥库文件zlex.keystore，对于加载密钥库密文件实现已在第10章中有详细描述，这里不再复述。

相信读者朋友对于HttpURLConnection这个类并不陌生，该类用于HTTP协议的访问。我们现在需要访问HTTPS协议，则需要HttpsURLConnection类。

HttpsURLConnection类与HttpURLConnection类差别不大，通过HttpsURLConnection类的setSSLSocketFactory()方法配置SSLSocektFactory实例就可将其作为一般HttpURLConnection类来操作。

SSLSocektFactory类用于管理SSL套接字，需要通过SSLContext类的getSocketFactory()方法获得。而SSLContext类初始化时则需要配置相应的密钥库与信任库。

在这里密钥库与信任库可以指向同一个文件，即zlex.keystore文件。我们可以将其加载配置SSLContext类的实例化对象。

获得密钥库管理工厂（KeyManagerFactory）如代码清单11-5所示。

代码清单11-5 获得密钥库管理工厂

```
// 实例化密钥库管理工厂
KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
// 获得密钥库
KeyStore keyStore = getKeyStore(keyStorePath, password);
// 初始化密钥库管理工厂
keyManagerFactory.init(keyStore, password.toCharArray());
```

上述getKeyStore()方法即加载密钥库方法，获得密钥库后就可初始化密钥库管理工厂。

接下来，我们需要获得信任库管理工厂（TrustManagerFactory），如代码清单11-6所示。

更多资源请访问稀酷客(www.ckook.com)

代码清单11-6 获得信任库管理工厂

```
// 实例化信任库管理工厂
TrustManagerFactory trustManagerFactory = TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
// 获得信任库
KeyStore trustStore = getKeyStore(trustStorePath, password);
// 初始化信任库管理工厂
trustManagerFactory.init(trustStore);
```

获得信任库实现与获得密钥库实现同出一辙，都是通过加载密钥库进行初始化操作。

完成上述操作后，我们就可以获得SSL上下文（SSLContext类），并获得SSL套接字工厂（SSLSocektFactory），如代码清单11-7所示。

代码清单11-7 获得SSL套接字工厂

```
// 实例化SSL上下文
SSLContext ctx = SSLContext.getInstance("TLS");
// 初始化SSL上下文
ctx.init(keyManagerFactory.getKeyManagers(), trustManagerFactory. getTrustManagers(),
new SecureRandom());
// 获得SSLSocketFactory
SSLSocketFactory socketFactory = ctx.getSocketFactory();
```

这里，我们实例化SSL上下文时使用的是TLS协议，也可使用SSL协议。

获得SSLSocketFactory实例化对象后，我们只需要通过HttpsURLConnection类的setSSLSocketFactory()方法进行配置即可。完整代码如代码清单11-8所示。

代码清单11-8 HttpsURLConnection配置

```
import java.io.FileInputStream;
import java.security.KeyStore;
import java.security.SecureRandom;
import javax.net.ssl.HttpsURLConnection;
import javax.net.ssl.KeyManagerFactory;
import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLSocketFactory;
import javax.net.ssl.TrustManagerFactory;
/**
 * HTTPS组件
 * @author 梁栋
 * @version 1.0
 */
public abstract class HTTPSCoder {
    /**
     * 协议
     * 支持TLS和SSL协议
     */
    public static final String PROTOCOL = "TLS";
    /**
```

更多资源请访问稀酷客(www.ckook.com)

```
     * 获得KeyStore
     * @param keyStorePath 密钥库路径
     * @param password 密码
     * @return KeyStore 密钥库
     * @throws Exception
     */
    private static KeyStore getKeyStore(String keyStorePath, String password)
    throws Exception {
            // 实例化密钥库
            KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());
            // 获得密钥库文件流
            FileInputStream is = new FileInputStream(keyStorePath);
            // 加载密钥库
            ks.load(is, password.toCharArray());
            // 关闭密钥库文件流
            is.close();
            return ks;
    }
    /**
     * 获得SSLSocektFactory
     * @param password 密码
     * @param keyStorePath 密钥库路径
     * @param trustStorePath 信任库路径
     * @return SSLSocketFactory
     * @throws Exception
     */
    private static SSLSocketFactory getSSLSocketFactory(String password, String
    keyStorePath, String trustStorePath) throws Exception {
            // 实例化密钥库
            KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance
            (KeyManagerFactory.getDefaultAlgorithm());
            // 获得密钥库
            KeyStore keyStore = getKeyStore(keyStorePath, password);
            // 初始化密钥工厂
            keyManagerFactory.init(keyStore, password.toCharArray());
            // 实例化信任库
            TrustManagerFactory trustManagerFactory =
            TrustManagerFactory.getInstance (TrustManagerFactory.
            getDefaultAlgorithm());
            // 获得信任库
            KeyStore trustStore = getKeyStore(trustStorePath, password);
            // 初始化信任库
            trustManagerFactory.init(trustStore);
            // 实例化SSL上下文
            SSLContext ctx = SSLContext.getInstance(PROTOCOL);
            // 初始化SSL上下文
            ctx.init(keyManagerFactory.getKeyManagers(), trustManagerFactory.
            getTrustManagers (), new SecureRandom());
```

更多资源请访问稀酷客(www.ckook.com)

```
            // 获得SSLSocketFactory
            return ctx.getSocketFactory();
        }
        /**
         * 为HttpsURLConnection配置SSLSocketFactory
         * @param conn HttpsURLConnection
         * @param password 密码
         * @param keyStorePath 密钥库路径
         * @param trustStorePath 信任库路径
         * @throws Exception
         */
        public static void configSSLSocketFactory(HttpsURLConnection conn, String
        password, String keyStorePath, String trustStorePath) throws Exception {
            // 获得SSLSocketFactory
            SSLSocketFactory sslSocketFactory = getSSLSocketFactory(password,
            keyStorePath, trustStorePath);
            // 设置SSLSocketFactory
            conn.setSSLSocketFactory(sslSocketFactory);
        }
    }
```

如果HTTPS连接建立失败会怎样？我们将不能从连接中获得有效的ContentLength，即ContentLength值为-1。根据这一特性，我们可以鉴别HTTPS连接是否成功，并验证是否可以获得内容，相关实现如代码清单11-9所示。

代码清单11-9 获取HTTPS连接内容

```
// 鉴别内容长度
int length = conn.getContentLength();
byte[] data = null;
// 如果内容长度为-1，则放弃解析
if (length != -1) {
    DataInputStream dis = new DataInputStream(conn.getInputStream());
    data = new byte[length];
    dis.readFully(data);
    dis.close();
    System.err.println(new String(data));
}
// 关闭连接
conn.disconnect();
```

如果HTTPS连接成功，我们可以供输入流中获得相应的信息，并且该信息已经过解密，我们可以在控制台的到相应的内容。

完整的测试用例如代码清单11-10所示。

代码清单11-10 HTTPS连接测试

```
import static org.junit.Assert.*;
import java.io.DataInputStream;
```

更多资源请访问稀酷客(www.ckook.com)

```
import java.net.URL;
import javax.net.ssl.HttpsURLConnection;
import org.junit.Test;
/**
 * HTTPS测试
 * @author 梁栋
 * @version 1.0
 */
public class HTTPSCoderTest {
    // 密钥库/信任库密码
    private String password = "123456";
    // 密钥库文件路径
    private String keyStorePath = "d:/zlex.keystore";
    // 信任库文件路径
    private String trustStorePath = "d:/zlex.keystore";
    // 访问地址
    private String httpsUrl = "https://www.zlex.org/ssl/";
    /**
     * HTTPS验证
     * @throws Exception
     */
    @Test
    public void test() throws Exception {
        // 建立HTTPS链接
        URL url = new URL(httpsUrl);
        HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
        // 打开输入输出流
        conn.setDoInput(true);
        // 为HttpsURLConnection配置SSLSocketFactory
        HTTPSCoder.configSSLSocketFactory(conn, password, keyStorePath, trustStorePath);
        // 鉴别内容长度
        int length = conn.getContentLength();
        byte[] data = null;
        // 如果内容长度为-1，则放弃解析
        if (length != -1) {
            DataInputStream dis = new DataInputStream(conn.getInputStream());
            data = new byte[length];
            dis.readFully(data);
            dis.close();
            System.err.println(new String(data));
        }
        // 关闭连接
        conn.disconnect();
        // 验证
        assertNotNull(data);
    }
}
```

更多资源请访问稀酷客(www.ckook.com)

通过代码访问https://www.zlex.org/ssl/页面，我们在控制台得到request相关属性与通过浏览器访问获得的内容有所不同。除了javax.servlet.request.ssl_session属性必然发生变化外，javax.servlet.request.cipher_suite属性也发生了变化。这里javax.servlet.request.cipher_suite属性值为“SSL_RSA_WITH_RC4_128_MD5”，而非“TLS_RSA_WITH_AES_128_CBC_SHA”。这说明不同的客户端在同一时间访问同一服务时，将有可能使用不同的协议或算法。

在上述代码中，我们并没有做过任何加密/解密操作。我们像使用HttpURLConnection类一样使用HttpsURLConnection类构建连接，我们无需关心加密/解密的具体实现。这些加密/解密的实现层位于传输层，对于应用层完全透明，这极大地方便了我们的使用。

11.4 双向认证服务

双向认证服务与单向认证服务的主要差别在于双向认证服务增加了客户证书验证环节。这使得消息收发双方可以通过证书相互验证对方的身份，达到双向认证的作用。

双向认证服务需要根证书、服务器证书和客户证书共3项证书。我们可以通过OpenSSL工具构建这3项证书对应的个人信息交换文件（ca.p12、server.p12和client.p12文件）。

为便于演示，本文使用OpenSSL工具构建自签名根证书、服务器证书和客户证书，搭建双向认证服务。实际应用时，请读者朋友使用经由CA机构签发的根证书。

本文将使用ca.p12、server.p12和client.p12个人信息交换文件构建双向认证，个人信息交换文件构建相关实现请参考第10章相关内容。

11.4.1 准备工作

双向认证服务实现同样不需要实现任何代码，与单向认证服务构建几乎毫无差别。

由于这里我们使用域名www.zlex.org作为用户姓名，因此仍需进行域名绑定，相关操作可参照11.3节的内容。除此之外，我们需要导入根证书和客户证书，同时需要对Tomcat做相应修改。

1. 证书导入

本文将使用ca.p12、server.p12和client.p12个人信息交换文件构建双向认证，在验证操作之前，请导入ca.p12和client.p12文件。

❑ 导入ca.p12文件

ca.p12文件是CA根证书的个人信息交换文件，我们可以像导入zlex.cer文件那样将其导入。需要注意的是，这里我们导入的是个人信息交换文件，在“证书导入向导”对话框中点击“浏览”按钮时，指定个人信息交换文件格式（*.p12;*.pfx），如图11-22所示。

选择ca.p12文件，单击“下一步”按钮，在密码输入框中输入CA根证书密码（这里为“123456”），如图11-23所示。

单击下一步按钮，直至证书导入成功。最终，我们将在“受信任的根证书颁发机构”选项卡中找到我们导入的自签名证书，如图11-24所示。

更多资源请访问稀酷客(www.ckook.com)

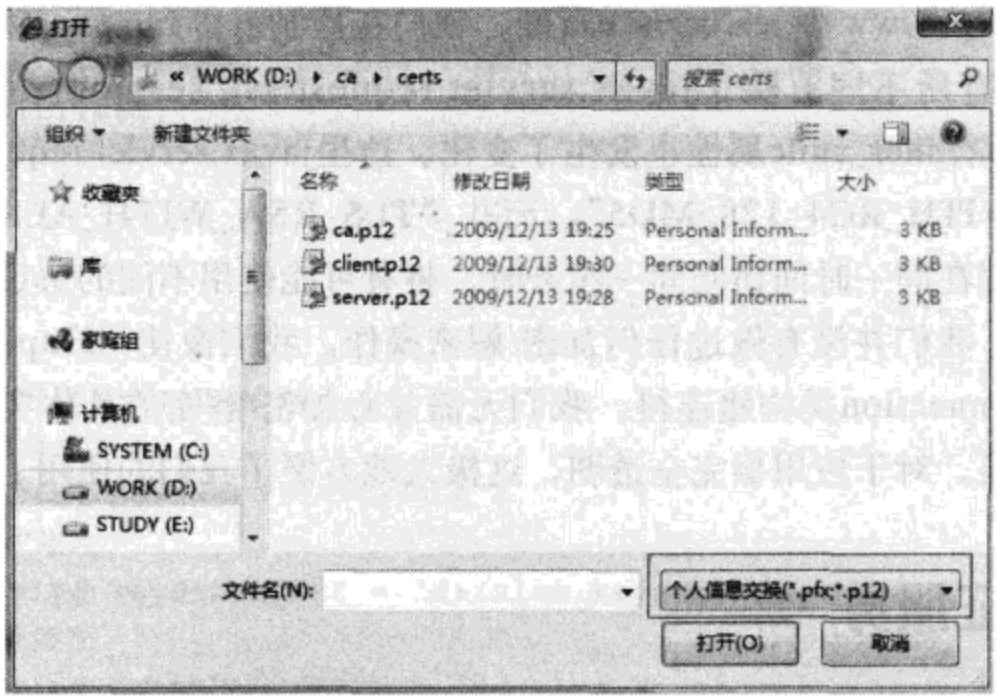


图11-22 选择证书导入

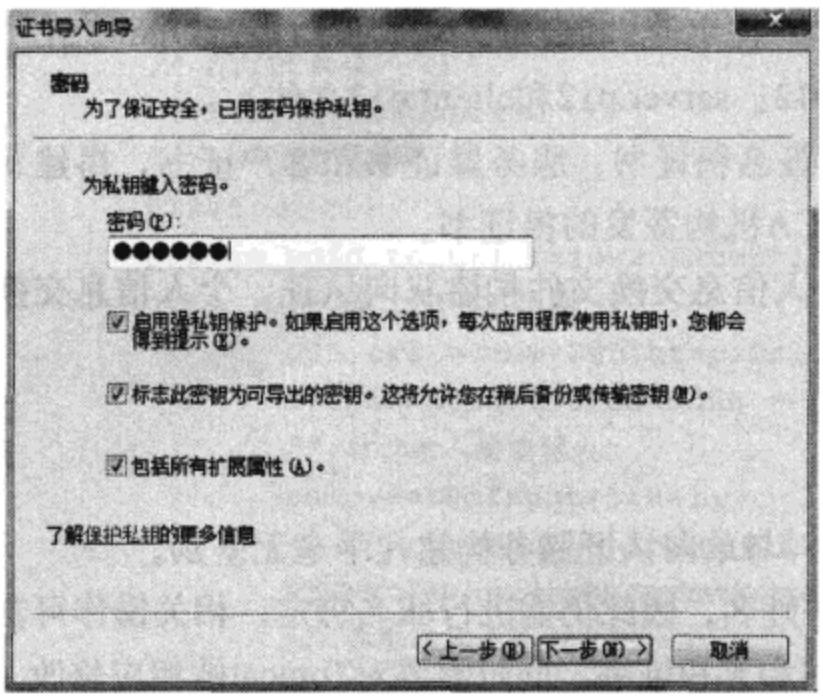


图11-23 输入密码



图11-24 导入自签名根证书2

❑ 导入client.p12文件

client.p12文件导入与ca.p12文件导入基本相同。

在“证书”对话框中选择“个人”选项卡，点击“导入”按钮。在“证书导入向导”对话框中点击“浏览”按钮时，指定个人信息交换文件格式（*.p12;*.pfx）。选择client.p12文件，单击“下一步”按钮，在密码输入框中输入客户证书密码（这里为“123456”），单击下一步按钮，直至证书导入成功。最终，我们将在“个人”选项卡中找到我们导入的客户证书，如图11-25所示。

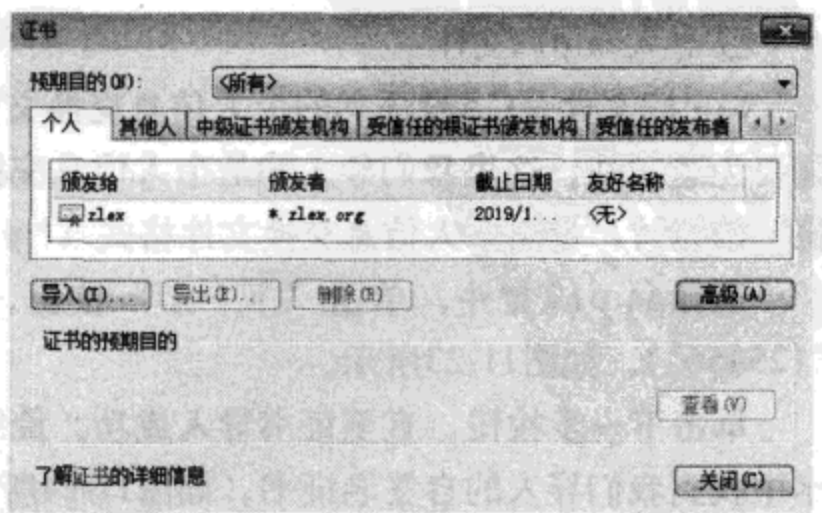


图11-25 导入客户证书

更多资源请访问稀酷客(www.ckook.com)

2. 服务器配置

这里我们需要将ca.p12和server.p12文件复制到Tomcat的conf目录下。同时，需要在Windows系统下通过IE浏览器导入ca.p12和client.p12文件。

接下来，我们需要在Tomcat的server.xml文件中配置双向认证，如代码清单11-11所示。

代码清单11-11 配置SSL/TLS双向认证

```
<Connector
    SSLEnabled="true"
    clientAuth="true"
    maxThreads="150"
    port="443"
    protocol="HTTP/1.1"
    scheme="https"
    secure="true"
    sslProtocol="TLS"
    keystoreFile="conf/server.p12"
    keystorePass="123456"
    keystoreType="PKCS12"
    truststoreFile="conf/ca.p12"
    truststorePass="123456"
    truststoreType="PKCS12" />
```

这里的密钥库文件参数keystoreFile指向server.p12文件，密钥库密码参数keystorePass值为“123456”，密钥库类型参数keystoreType值为“PKCS12”。

这里请读者朋友注意信任库的参数配置。与单向认证服务配置不同，双向认证服务区分信任库文件和密钥库文件。此时，server.p12文件将作为密钥库文件，而ca.p12文件则作为信任库文件。

因此，信任库文件参数truststoreFile指向ca.p12文件，信任库密码参数truststorePass值为“123456”，信任库类型参数truststoreType值为“PKCS12”。这一点与单向认证有着明显的差别，往往容易被忽视，将信任库与密钥库混为一谈。

在上述配置中，我们将客户端验证参数clientAuth的值置为“true”，这是打开双向认证的关键一步。

现在，我们在浏览器中访问地址https://www.zlex.org/，结果如图11-26所示。

这说明双向认证已经成功构建，此处浏览器要求用户确认使用客户证书。“单击此处查看证书属性”，我们将获得客户证书“常规”信息，如图11-27所示。

点击图11-26中的“确定”按钮将给出密钥授权提示，如图11-28所示。

在“申请使用密钥的权限”对话框中选中“授予权限”并确认后，我们将得到如图11-14所示的Tomcat欢迎页面。

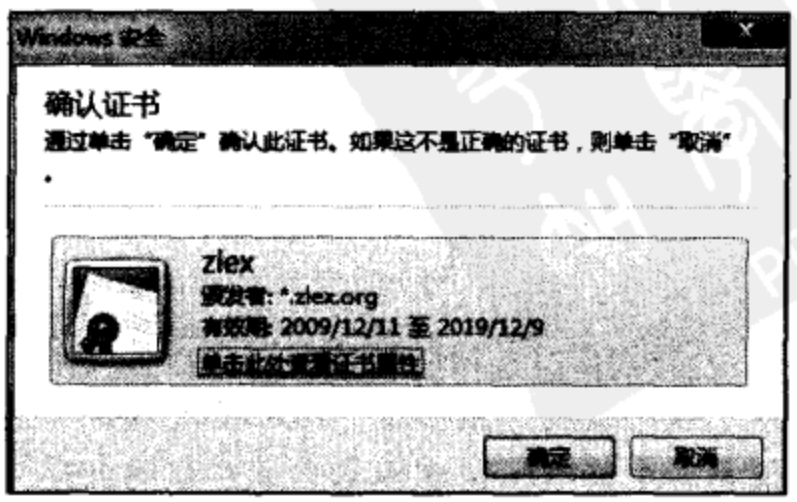


图11-26 客户证书确认提示

更多资源请访问稀酷客(www.ckook.com)



图11-27 客户证书

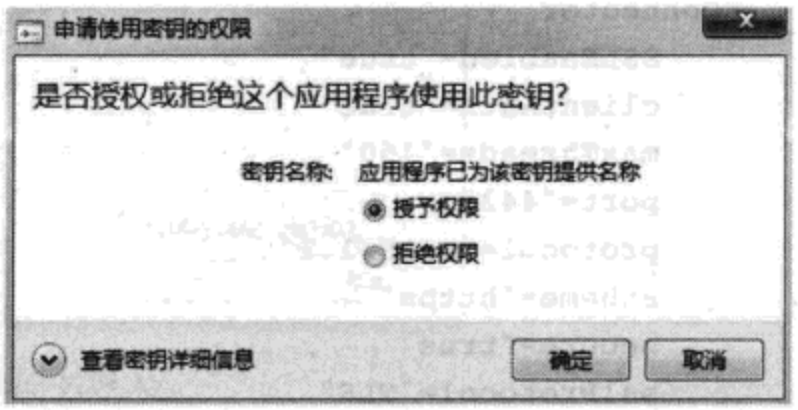


图11-28 密钥授权提示

其实，我们在使用一些电子商务平台时也会使用到双向认证服务器，但却从未见到如图11-26和图11-28所示的页面。这是因为电子商务平台为了提高用户使用便捷性，通过ActiveX控件替用户完成了这些页面的相关操作。

如果我们未能导入client.p12文件，浏览器则会提示该页面访问失败。

11.4.2 服务验证

我们继续访问地址https://www.zlex.org/ssl/，这时将获得不一样的信息，如图11-29所示。

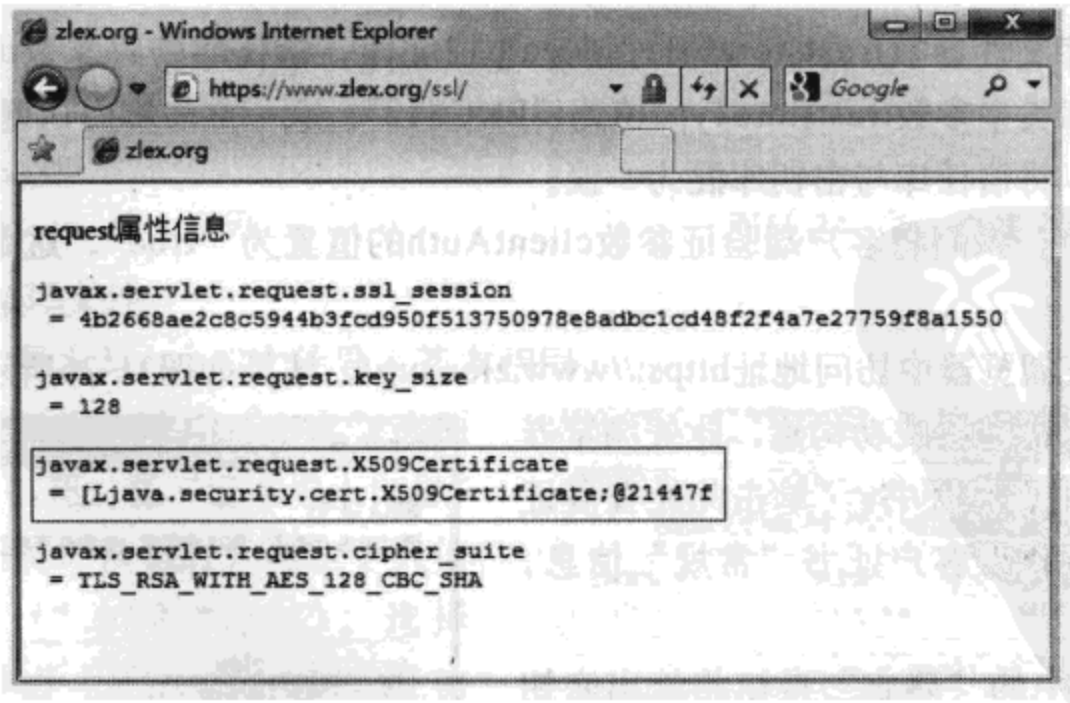


图11-29 SSL/TLS协议中的request属性2

我们发现这里多了一项request属性——“javax.servlet.request.X509Certificate”，并且属性值是一个数组，该属性指向客户证书列表。

我们需要对index.jsp页面做细微调整，将该证书的相关信息打印到页面上。完整实现如代

更多资源请访问稀酷客(www.ckook.com)

码清单11-12所示。

代码清单11-12 index.jsp页面2

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ page import="java.util.Enumeration,java.security.cert.X509Certificate"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>zlex.org</title>
</head>
<body>
<p>request属性信息</p>
<pre>
<%
    for (Enumeration en = request.getAttributeNames(); en.hasMoreElements();) {
            String name = (String) en.nextElement();
            out.println(name);
            out.println(" = " + request.getAttribute(name));
            out.println();
    }
%>
</pre>
<p>数字证书信息</p>
<pre>
<%
    X509Certificate[] certs = (X509Certificate[]) request.getAttribute
    ("javax.servlet.request.X509Certificate");
    for (X509Certificate cert : certs) {
            out.println("版本: \t\t" + cert.getVersion());
            out.println("序列号: \t\t" + cert.getSerialNumber());
            out.println("颁布者: \t\t" + cert.getIssuerDN().getName());
            out.println("使用者: \t\t" + cert.getSubjectDN().getName());
            out.println("签名算法: \t" + cert.getSigAlgName());
            out.println("证书类型: \t" + cert.getType());
            out.println("有效期从: \t" + cert.getNotBefore());
            out.println("至: \t\t" + cert.getNotAfter());
    }
%>
</pre>
</body>
</html>
```

我们再次访问地址https://www.zlex.org/ssl/，将获得客户证书信息，如图11-30所示。

很显然，浏览器在访问该服务时，在request属性中载入客户证书。如果我们通过代码默认

更多资源请访问稀酷客(www.ckook.com)

浏览器访问时，需要加载客户证书。



图11-30 SSL/TLS协议中的request属性和客户证书信息

11.4.3 代码验证

由于我们使用PKCS#12格式的个人信息交换文件作为密钥库和信任库文件，所以，我们需要对原有代码稍作调整。

首先，需要调整getKeyStore()方法，修改实例化密钥库参数，如代码清单11-13所示。

代码清单11-13 修改获取密钥库方法

```
private static KeyStore getKeyStore(String keyStorePath, String password) throws Exception {
    // 实例化密钥库
    KeyStore ks =  KeyStore.getInstance("PKCS12");
    //省略
}
```

其次，我们需要对测试用例代码做修改，将密钥库和信任库分别指向server.p12和ca.p12文件。如代码清单11-14所示。

代码清单11-14 修改密钥库和信任库文件路径

```
public class HTTPSCoderTest {
    //省略
    // 密钥库文件路径
    private String keyStorePath = "d:/server.p12";
```

更多资源请访问稀酷客(www.ckook.com)

```
        // 信任库文件路径
        private String trustStorePath = "d:/ca.p12";
        //省略
    }
```

11.5 应用举例

随着电子商务的快速发展，网络银行成为网上交易的必要平台。工商银行、农业银行、中国银行、建设银行、交通银行五大国有商业银行都在其官网上开通了网银交易业务。深圳发展银行、中信银行、民生银行、华夏银行、招商银行、广东发展银行、兴业银行、光大银行等股份制银行也相继在其官网上开通了网银交易业务。如何确保网络交易安全成为各网络银行不容忽视的第一要务。

经作者一番调查后发现，所有网银系统都采用了SSL/TLS证书加密技术，完全支持单向认证服务。绝大多数网银系统甚至采用了客户证书，可以支持双向认证服务。

网银系统通常提供大众版（普及版）和专业版两种登录方式。大众版（普及版）登录采用为安全密码控件（ActiveX控件）防止用户输入密码时被间谍软件非法截获，可以达到单向认证的作用；专业版登录要求更为严格，强制要求用户使用客户证书登录系统，达到双向认证的作用。通常专业版登录配备USB Key （U盾）硬件数字证书设备，通过硬件上的程序对客户证书定期更新。

如果对本文提到的各大网银系统所使用的数字证书做详细调查，我们会发现几乎所有的网银系统都是用了由VeriSign（http://www.verisign.com/）签发的数字证书，为保证密钥安全其证书的有效期一般不会超过3年。

我们所熟知的支付宝（https://www.alipay.com/）同样采用了SSL/TLS证书加密技术构建的电子商务平台，并提供客户证书下载支持双向认证服务。

如今的网络应用已经跨越了不同技术架构之间的鸿沟，不同计算机语言（如Java和C#）实现的网络应用平台可以通过Web Service平台进行数据交互，甚至是方法级的相互调用。各应用程序通过网络协议和规定的一些标准数据格式（HTTP、XML、SOAP）来访问Web Service平台，通过Web Service平台内部执行得到所需结果。Web Service属于开放式异构体交互平台，请求和回复内容均以明文传递，如果不加以保护商业数据极容易在交互过程中被窃取。通常，Web Service平台交互双方使用SSL/TLS证书加密技术确保数据安全。

11.6 小结

HTTPS协议和SSL/TLS协议分属TCP/IP参考模型中的应用层和传输层。HTTPS协议是Web上最为常用的安全访问协议。简单地说，HTTPS就是HTTP安全版，HTTPS是基于SSL/TLS的HTTP协议，或者说HTTPS=SSL/TLS+HTTP。

SSL/TLS协议包含两个协议：SSL（Secure Socket Layer，安全套接字层）和TLS

更多资源请访问稀酷客(www.ckook.com)

（Transport Layer Security，传输层安全）协议。SSL共有3个版本：SSL1.0、SSL2.0和SSL3.0。IETF在基于SSL3.0协议的基础上发布了TLS1.0、TLS1.0与SSL3.0几乎是兼容的。

SSL/TLS协议通过使用数字证书确保网络交互安全，为数字证书的使用提供了最佳的应用环境。

基于HTTPS协议的网络平台需要通过底层的SSL/TLS协议完成协商算法、验证证书和产生密钥的工作，最后才能进行加密交互。通过HTTPS协议可以构建两种认证服务：单向认证服务和双向认证服务。单向认证服务仅提供服务器身份验证，即提供服务器证书；双向认证要求服务器提供证书的同时，强制客户端提供证书，最终达到服务器和客户端双方身份的验证。

为确保网络交易安全，各大网银系统均使用HTTPS协议保护数据安全。通常，这些网银系统使用的数字证书有效期不超过3年。为确保交易安全，网银系统提供USB Key（U盾）硬件数字证书，最终起到双向认证的作用。

搭建HTTPS服务比我们设想的要更为简单，以Tomcat配置HTTPS服务为例，通过简单配置即可构建单向认证服务和双向认证服务。Java提供两种密钥库格式：JKS和PKCS12，需在配置时注意。

更多资源请访问稀酷客(www.ckook.com)

第12章

量体裁衣——为应用选择合适的装备

信息是这个时代最核心的内容，网络是这个时代最主要的载体，网络信息安全终于成为这一时代的主题。我们该如何确保论坛、博客、社区等开放性网络应用的用户隐私不被泄露？我们如何避免QQ、MSN、RTX、Skype、GTalk等聊天工具在传递敏感数据时不被窃听？我们该如何保证与合作伙伴交换的商业数据不被窃取？

本章将通过3套较为常见的实例，介绍如何为应用选择合适的加密算法，确保数据安全。

12.1 实例：常规Web应用开发安全

BBS、BLOG以及越来越火爆的SNS正一次又一次地“绑架”了我们的生活，我们正不遗余力地在这些“场所”投入我们的时间、精力甚至是汗水。我们关注每一条行业消息、用心发表每一篇帖子、细心维护自己的日志，哪怕半夜都要爬起来去“偷菜”。倘若有一天别人盗用了我们的账号，发表了不该有的言论、清空了我们的日志，或是变卖了我们的“家产”，我们长久以来积累起来的威信、声誉、财富等都荡然无存。如果言论涉及伦理道德或者法律等问题，这对于用户群体来说就是天大的麻烦。某些系统中保留了我们的私人信息，包括所在公司、住址、私人相册、电子邮箱、手机号码甚至是身份证号，一旦私密信息泄露，将给我们的生活带来很多不必要的麻烦。造成这一局面的主要根源不外乎账号管理问题！

很多系统在设计时都将密码以明文的方式存储在数据库中，并以此为基础提供取回密码的服务，这使得用户私密数据泄露成为可能。如果仅仅将密码以明文的方式存储在数据库中，任何拥有数据库访问权限的人都可以获得某个用户的访问权限。倘若开放取回密码的服务，任何具有该权限的人同样可以获得该用户的访问权限。通过使用该用户的账号，可以假借用户之名进行任何操作，谋取私利而不留痕迹。

本文以常规Web应用登录模块为例，介绍如何通过消息摘要算法隐蔽用户敏感信息。

12.1.1 常规Web应用基本实现

这里，我们将构建一个名为“SNS”的Web应用，并使用MySQL为SNS提供数据库服务。

更多资源请访问稀酷客(www.ckook.com)

1. 准备工作

本书将使用MySQL作为SNS应用数据库，并使用Commons Codec提供消息摘要算法实现。有关MySQL数据库下载及安装操作，请参考相关资料。同时，请读者注意下载MySQL JDBC包。有关Commons Codec使用，请参考第4章和第6章的内容。

2. 搭建数据库

首先，我们需要为SNS构建数据库。建库SQL如代码清单12-1所示。

代码清单12-1 SNS库SQL

```
CREATE DATABASE `sns`
DEFAULT CHARACTER SET utf8
```

接下来，我们需要构建用户表Account，主要包含主键（account_id字段）、用户名（name字段）、用户密码（password字段）和电子邮箱（email字段）共4项基本信息。建表SQL如代码清单12-2所示。

代码清单12-2 Account表SQL

```
CREATE TABLE  `sns`.`account` (
  `account_id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(45) NOT NULL,
  `password` varchar(45) NOT NULL,
  `email` varchar(45) NOT NULL,
  PRIMARY KEY (`account_id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

为方便演示，我们构建3条用户数据。插入用户信息SQL如代码清单12-3所示。

代码清单12-3 插入用户信息SQL

```
INSERT INTO `sns`.`account`(`name`,`password`,`email`) values('Admin','Admin','admin@zlex.org')
INSERT INTO `sns`.`account`(`name`,`password`,`email`) values('zlex','zlex','zlex@zlex.org')
INSERT INTO `sns`.`account`(`name`,`password`,`email`) values('snowolf','snowolf','snowolf@zlex.org')
```

上述SQL中，用户密码即用户名，并以明文存储，如图12-1所示。

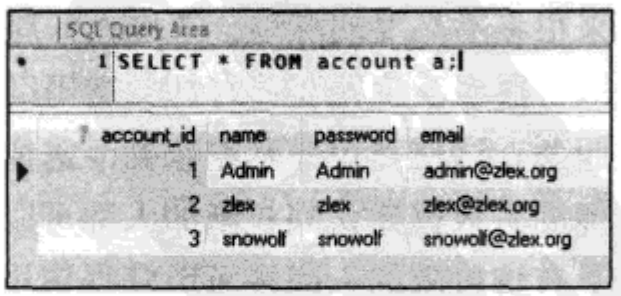


图12-1 检索用户数据1

任何拥有数据库访问权限的人都可以获得某个用户的访问权限，这是很不安全的。

3. 构建应用

完成数据库搭建操作后，我们将构建应用。这里我们构建一个简单的用户登录模块。

首先，我们需要构建web.xml文件，如代码清单12-4所示。

代码清单12-4 web.xml文件

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
```

更多资源请访问稀酷客(www.ckook.com)

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
        id="WebApp_ID"
        version="2.5">
        <display-name>web</display-name>
        <welcome-file-list>
            <welcome-file>index.jsp</welcome-file>
        </welcome-file-list>
    </web-app>
```

接下来，我们将构建两个页面：index.jsp和login.jsp。index.jsp页面给出用户登录入口，login.jsp页面操作数据库验证用户身份。

index.jsp页面代码如代码清单12-5所示。

代码清单12-5 index.jsp页面

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>欢迎</title>
</head>
<body>
<form action="login.jsp" method="post">
<table>
    <tr>
            <td>username:</td>
            <td><input name="username" /></td>
    </tr>
    <tr>
            <td>password:</td>
            <td><input name="password" type="password" /></td>
    </tr>
    <tr>
            <td colspan="2"><input type="submit" value="登录" />
                <input type="reset" value="重置" /></td>
    </tr>
</table>
</form>
</body>
</html>
```

index.jsp页面与一般的HTML页面几乎没有任何差别，仅仅用于展示用户登录入口。

更多资源请访问稀酷客(www.ckook.com)

login.jsp页面用于处理用户登录验证逻辑，如代码清单12-6所示。

代码清单12-6 login.jsp页面1

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ page import="java.sql.*"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>登录结果</title>
</head>
<body>
<%
    // 构建数据库连接
    // JDBC驱动类
    String driverClass = "com.mysql.jdbc.Driver";
    // 用户名
    String name = "root";
    // 密码
    String pwd = "admin";
    // 数据库连接
    String url = "jdbc:mysql://localhost:3306/sns";
    Class.forName(driverClass);
    Connection conn = DriverManager.getConnection(url, name, pwd);
    // 取得表单信息
    String username = request.getParameter("username");
    String password = request.getParameter("password");
    // 检索用户
    PreparedStatement statement = conn.prepareStatement("select * from account
    where name = ?");
    statement.setString(1, username);
    ResultSet rs = statement.executeQuery();
%>
<pre>
<%
    // 如果记录存在验证用户身份
    if (rs.next()) {
        String p = rs.getString("password");
        if (p.equals(password)) {
            out.println("用户 " + username);
            out.println("登录成功! ");
        } else {
            out.println("用户 " + username);
            out.println("密码错误! ");
        }
    } else {
```

更多资源请访问稀酷客(www.ckook.com)

```
            out.println("用户 " + username + " 不存在");
        }
    %>
    </pre>
    <%
        // 关闭结果集
        rs.close();
        // 关闭语句集
        statement.close();
        // 关闭连接
        conn.close();
    %>
    </body>
    </html>
```

用户验证的逻辑很简单，按用户名检索用户信息。如果用户记录存在，则校验用户密码。如果密码校验通过，则用户登录成功。当然，我们也可以按用户名和密码检索用户信息。

4. 验证服务

启动Tomcat，访问地址http://localhost:8080/sns/，登录页面如图12-2所示。

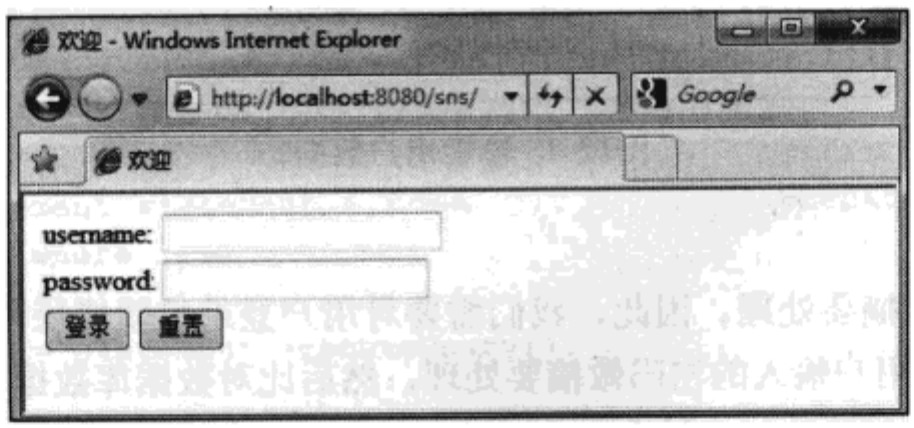


图12-2 登录页面

输入用户名“Admin”和密码“Admin”，点击“登录”按钮，我们将成功登录该系统，如图12-3所示。

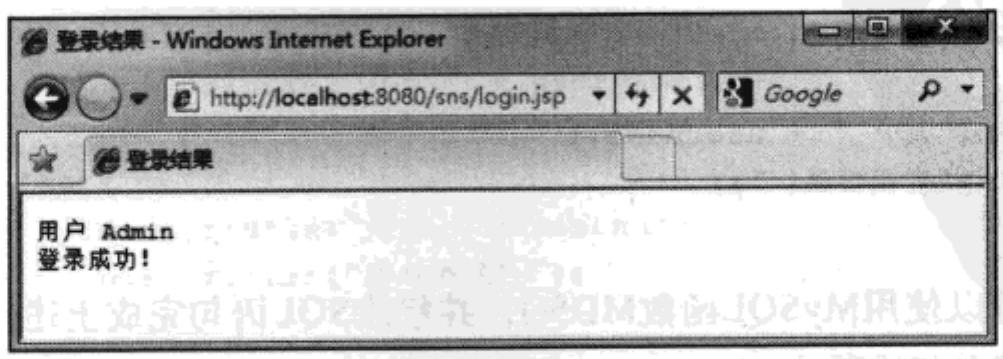


图12-3 登录成功页面

这是一个再简单不过的用户登录验证模块，相信很多读者对此都非常熟悉。当然，这样的系统并不安全。这里的“安全”针对拥有数据库访问权限的任何人，甚至是这套系统的开发者。用户敏感数据必须加以隐蔽，避免他人盗用。此处，我们需要隐蔽用户的密码信息，使用消息摘要算法隐蔽用户密码。

更多资源请访问稀酷客(www.ckook.com)

12.1.2 安全升级1——摘要处理

针对这样的系统，我们可以使用摘要算法对其进行安全加固，确保用户敏感数据不被泄露。

1. 修改用户数据

MySQL提供了MD5和SHA1两种消息摘要算法函数：MD5()和SHA1()。我们需要对数据库稍作调整，使用MD5算法隐藏用户密码（这里用户密码为用户名），更新用户密码SQL如代码清单12-7所示。

代码清单12-7 更新用户密码SQL1

```
UPDATE `account`
SET `password` = MD5(`name`)
```

这里，我们使用了MD5算法函数MD5()。再次检索用户表Account，用户密码已经变得“面目全非”了，这正是我们的目的！如图12-4所示。

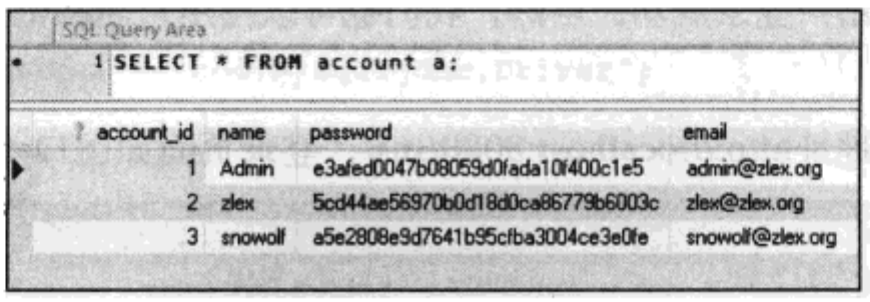


图12-4 检索用户数据2

2. 修改校验模块

用户密码已经经过摘要处理，因此，我们需要对用户登录校验模块稍作修改。这里，我们使用Common Codec对用户输入的密码做摘要处理，然后比对数据库数据。对原login.jsp页面校验部分稍作调整，如代码清单12-8所示。

代码清单12-8 校验用户密码1

```
if (p.equals(DigestUtils.md5Hex(password))) {
    out.println("用户 " + username);
    out.println("登录成功！");
} else {
    out.println("用户 " + username);
    out.println("密码错误！");
}
```

当然，我们也可以使用MySQL函数MD5()，并修改SQL语句完成上述操作。login.jsp页面的完整代码如代码清单12-9所示。

代码清单12-9 login.jsp页面2

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ page import="org.apache.commons.codec.digest.DigestUtils"%>
<%@ page import="java.sql.*"%>
```

更多资源请访问稀酷客(www.ckook.com)

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>登录结果</title>
</head>
<body>
<%
    // 构建数据库连接
    // JDBC驱动类
    String driverClass = "com.mysql.jdbc.Driver";
    // 用户名
    String name = "root";
    // 密码
    String pwd = "admin";
    // 数据库连接
    String url = "jdbc:mysql://localhost:3306/test";
    Class.forName(driverClass);
    Connection conn = DriverManager.getConnection(url, name, pwd);
    // 取得表单信息
    String username = request.getParameter("username");
    String password = request.getParameter("password");
    // 检索用户
    PreparedStatement statement = conn                    .prepareStatement("select *
    from account where name = ?");
    statement.setString(1, username);
    ResultSet rs = statement.executeQuery();
%>
<pre>
<%
    // 如果记录存在验证用户身份
    if (rs.next()) {
            String p = rs.getString("password");
            if (p.equals(DigestUtils.md5Hex(password))) {
                    out.println("用户 " + username);
                    out.println("登录成功! ");
            } else {
                    out.println("用户 " + username);
                    out.println("密码错误! ");
            }
    } else {
            out.println("用户 " + username + " 不存在");
    }
%>
</pre>
<%
    // 关闭结果集
```

更多资源请访问稀酷客(www.ckook.com)

```
        rs.close();
        // 关闭语句集
        statement.close();
        // 关闭连接
        conn.close();
    %>
    </body>
    </html>
```

如果我们对一些成熟系统仔细研究，就会发现这种使用摘要算法隐蔽用户密码的方法已经很普遍了。

对于上述代码修改的服务验证甚为简单，请读者自行验证。

12.1.3 安全升级2——加盐处理

常用的消息摘要算法如MD5和SHA1是现代主流计算机语言广泛支持的算法，在绝大多数数据库系统中都能找到相应的函数支持。若直观摘要值，单凭摘要值长度，我们就能判断出该系统大概是使用什么类型的摘要算法。MD5算法的摘要值为32位十六进制字符串，SHA1算法的摘要值为40位十六进制字符串。

借用香农的话“敌人知道系统”，这样的算法很容易被他人辨别。访问数据库的人很容易调用相关函数修改用户密码。为了加大数据破译难度，我们可以更改摘要算法，如使用摘要值长度更长的算法SHA224、SHA512等，很多数据库并未为这些算法提供支持。或者，我们可以换一种思路，对用户密码做混淆处理——加盐处理。

所谓加盐处理，就是在原有材料中加入其他成分（盐），增加系统复杂度。对于我们现在讨论的话题，“盐”必须是一种用户自有且不可变的元素。

我们可以假定系统要求用户有固定的用户名、电子邮箱等信息，这些信息完全可以作为“盐”与用户密码相结合，经过摘要处理后获得隐蔽性更强的摘要值。

1. 修改用户数据

为增强用户密码隐蔽性，我们使用用户的不可变信息——电子邮件作为我们所需要的“盐”，将其与用户原始密码（这里为用户名）做简单的连接，并使用SHA1算法对其做摘要处理，更新用户密码SQL如代码清单12-10所示。

代码清单12-10 更新用户密码SQL2

```
UPDATE `account`
SET `password` = SHA(CONCAT(`email`, `name`))
```

这里我们使用了SHA1算法函数SHA()，此外使用CONCAT()函数将用户的电子邮件和原有密码（这里的用户密码即是用户名）做了简单的连接。检索用户表Account，如图12-5所示。

如图12-5所示，通过密码长度，我们完全可以确定该密码经过SHA1算法处理。如果访问数据库的人对该系统并不了解，就根本无法破译或更改用户密码。

更多资源请访问稀酷客(www.ckook.com)

SQL Query Area

```
1 SELECT * FROM account a;
```

account_id	name	password	email
1	Admin	6ce9e5ea1931548cdb8290c9431bab80f3a2aalf	admin@zlex.org
2	zlex	ff61cd9b360a435d6b02bd23e7e45d19293e395b	zlex@zlex.org
3	snowolf	5ec831586568d0c3b9bac6e4b6e1f106d92a816c	snowolf@zlex.org

图12-5 检索用户数据3

2. 修改校验模块

我们再次对用户登录校验模块稍作修改，使用Common Codec对数据库中的用户电子邮箱以及用户输入的密码经连接后做摘要处理，然后比对数据库中的用户密码。对原login.jsp页面校验部分稍作调整，如代码清单12-11所示。

代码清单12-11 校验用户密码2

```
String email = rs.getString("email");
if (p.equals(DigestUtils.shaHex(email + password))) {
    out.println("用户 " + username);
    out.println("登录成功! ");
} else {
    out.println("用户 " + username);
    out.println("密码错误! ");
}
```

将用户电子邮箱同用户密码做简单的字符串连接是一种最为简单的加盐处理方式。我们完全可以在这个基础上做相应调整，使用其他算法加大破译难度。login.jsp页面完整代码如代码清单12-12所示。

代码清单12-12 login.jsp页面3

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ page import="org.apache.commons.codec.digest.DigestUtils"%>
<%@ page import="java.sql.*"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>登录结果</title>
</head>
<body>
<%
    // 构建数据库连接
    // JDBC驱动类
    String driverClass = "com.mysql.jdbc.Driver";
    // 用户名
    String name = "root";
```

更多资源请访问稀酷客(www.ckook.com)

```
        // 密码
        String pwd = "admin";
        // 数据库连接
        String url = "jdbc:mysql://localhost:3306/sns";
        Class.forName(driverClass);
        Connection conn = DriverManager.getConnection(url, name, pwd);
        // 取得表单信息
        String username = request.getParameter("username");
        String password = request.getParameter("password");
        // 检索用户
        PreparedStatement statement = conn
                .prepareStatement("select * from account where name = ?");
        statement.setString(1, username);
        ResultSet rs = statement.executeQuery();
%>
<pre>
<%
        // 如果记录存在验证用户身份
        if (rs.next()) {
                String p = rs.getString("password");
                String email = rs.getString("email");
                if (p.equals(DigestUtils.shaHex(email + password)))
{
                        out.println("用户 " + username);
                        out.println("登录成功! ");
                } else {
                        out.println("用户 " + username);
                        out.println("密码错误! ");
                }
        } else {
                out.println("用户 " + username + " 不存在");
        }
%>
</pre>
<%
        // 关闭结果集
        rs.close();
        // 关闭语句集
        statement.close();
        // 关闭连接
        conn.close();
%>
</body>
</html>
```

对于上述代码修改的服务验证，请读者自行验证。

相信读者对Spring Security框架有所了解，该框架提供了本书所提到的摘要处理和加盐处理两种实现，有兴趣的读者不妨一试。

更多资源请访问稀酷客(www.ckook.com)

12.2 实例：IM应用开发安全

大家在日常工作、生活中都有使用QQ、MSN、RTX、Skype、GTalk等IM（Instant Message，即时通信）工具的经历。换言之，缺少了QQ、MSN、RTX、Skype、GTalk的网络如同脱离了网络的电脑，变得索然无味。

IM工具与我们的工作和生活几乎是密不可分的，我们常常通过IM工具与亲友交流感情，与同事交换意见，甚至与合作伙伴互换商业数据。现如今的淘宝网生意做得越来越火，我们已经可以足不出户就在网上下订单，通过阿里旺旺跟卖家/买家讨价还价，享受网络生活的便利。

但是，当我们使用各种IM工具与对方交换数据时，可曾想过我们的聊天信息可能被监听，与合作伙伴交换的数据可能被窃取，刚刚通过阿里旺旺收到的手机充值卡号居然在瞬间被盗用。一切皆有可能，因为通过IM工具泄露银行卡号、密码的事情已经是不争的事实。

虽然这些IM工具都有自定的通信协议，但这些协议又必须公开，因此协议本身几乎没有任何安全性可言。尽管这些IM工具对自身不断进行升级，但极少数IM工具会对网络数据进行加密。因此，网络聊天通常都是不安全的。

QQ、MSN、RTX、Skype、GTalk等IM工具均基于UDP（User Datagram Protocol，用户数据报协议）进行通信。UDP是OSI 参考模型中一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务。相关细节请读者参考文档RFC 768（http://www.ietf.org/rfc/rfc768.txt）。

本文将构建简单的基于UDP的聊天工具，展示如何使用加密算法对聊天信息进行加密。

12.2.1 IM应用开发基本实现

相信大家对如图12-6所示的聊天窗口一定不会陌生。这是一个使用Java实现的基于UDP通信协议的聊天工具（我们称它为UDPChat）。当然，我们生活中使用的IM工具远比这个要复杂得多。接下来，我们将构建这样一个简单的桌面应用。

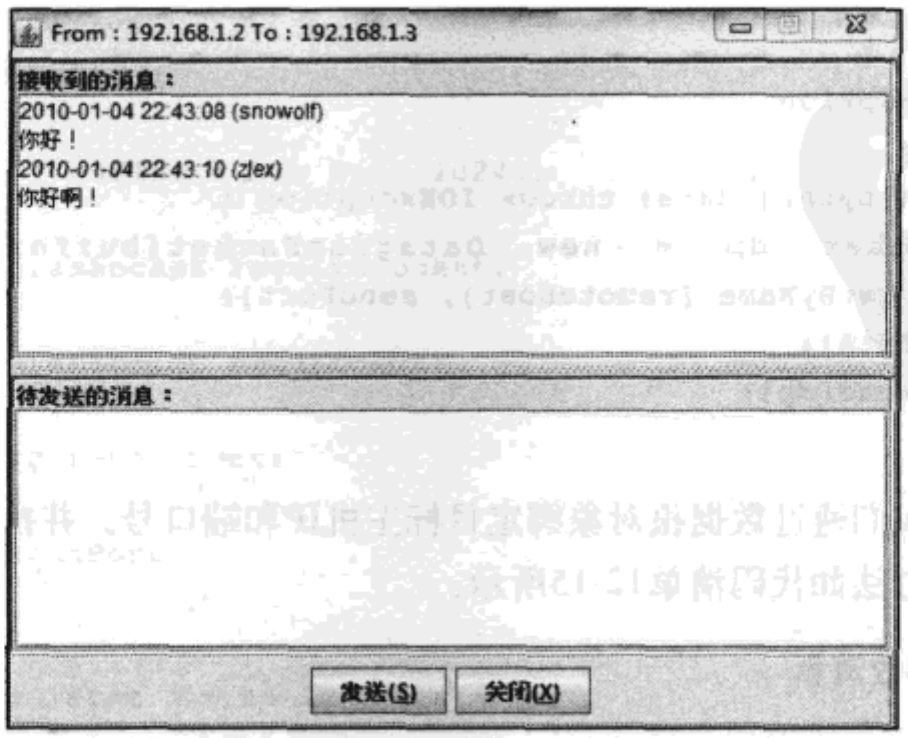


图12-6 UDP聊天窗口1

更多资源请访问稀酷客(www.ckook.com)

请读者在阅读本节内容之前了解与UDP相关的Java API，以及与Java Swing相关的API。本书将不对这些内容做相关细节介绍。

1. 构建应用

我们将构建UDPSocket、InitDialog和MainFrame 3个类，分别用于UDP通信协议交互、连接初始化和GUI实现。

❑ 构建UDPSocket

首先，我们需要构建UDPSocket类，用于控制UDP通信协议的交互。

这里，我们构建了两个用于数据报通信的套接字实例化对象：sendSocket和receiveSocket。sendSocket用于发送消息，receiveSocket用于接收消息，并将发送端口与接收端口分离，通过构造函数提供配置入口。这么做的目的是使得该桌面应用同时具有客户端和服务器端的通信功能。在现实生活中，我们真正使用的QQ、MSN等基本上都是通过一个固定端口与服务器进行交互。客户端轮询请求服务器端，获取消息。

在初始化数据报套接字时，需要注意将数据报接收套接字绑定在本机指定的端口上，而用于数据报发送的套接字则无需绑定。初始化数据报套接字如代码清单12-13所示。

代码清单12-13 初始化数据报套接字

```
// 初始化套接字
this.receiveSocket = new DatagramSocket(new InetSocketAddress(localHost, receivePort));
this.sendSocket = new DatagramSocket();
```

这里，我们将发送消息和接收消息分为两个方法：send()和receive()。

用于发送消息的方法如代码清单12-14所示：

代码清单12-14 发送消息

```
/**
 * 发送消息
 * @param data 消息
 * @throws IOException
 */
public void send(byte[] data) throws IOException {
    DatagramPacket dp = new DatagramPacket(buffer, buffer.length,
    InetAddress.getByName (remoteHost), sendPort);
    dp.setData(data);
    sendSocket.send(dp);
}
```

在上述方法中，我们通过数据报对象绑定目标主机IP和端口号，并执行发送操作。

用于接收消息的方法如代码清单12-15所示：

代码清单12-15 接收消息

```
/**
 * 接收消息
```

更多资源请访问稀酷客(www.ckook.com)

```
     * @return byte[] 消息
     * @throws IOException
     */
    public byte[] receive() throws IOException {
        DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
        receiveSocket.receive(dp);
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        baos.write(dp.getData(), 0, dp.getLength());
        byte[] data = baos.toByteArray();
        baos.flush();
        baos.close();
        return data;
    }
```

与消息发送方法不同，用于消息接收的方法无须在数据报对象中绑定目标主机IP和端口号。我们在初始化receiveSocket对象时，已经将其绑定在本机的指定端口号。

完整代码如代码清单12-16所示。

代码清单12-16　UDPSocket

```
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.net.SocketException;
/**
 * UDP套接字
 * @author 梁栋
 * @since 1.0
 */
public class UDPSocket {
    // 缓冲
    private byte[] buffer = new byte[1024];
    // 接收套接字
    private DatagramSocket receiveSocket;
    // 发送套接字
    private DatagramSocket sendSocket;
    // 目标主机
    private String remoteHost;
    // 发送端口
    private int sendPort;
    /**
     * 初始化
     * @param localHost 本地主机IP
     * @param remoteHost 远程主机IP
     * @param receivePort 接收端口
```

更多资源请访问稀酷客(www.ckook.com)

```
     * @param sendPort 发送端口
     * @throws SocketException
     */
    public UDPSocket(String localHost, String remoteHost, int receivePort, int
        sendPort) throws SocketException {
            this.remoteHost = remoteHost;
            this.sendPort = sendPort;
            // 初始化套接字
            this.receiveSocket = new DatagramSocket(new InetSocketAddress
            (localHost, receivePort));
            this.sendSocket = new DatagramSocket();
    }
    /**
     * 接收消息
     * @return byte[] 消息
     * @throws IOException
     */
    public byte[] receive() throws IOException {
            DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
            receiveSocket.receive(dp);
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            baos.write(dp.getData(), 0, dp.getLength());
            byte[] data = baos.toByteArray();
            baos.flush();
            baos.close();
            return data;
    }
    /**
     * 发送消息
     * @param data 消息
     * @throws IOException
     */
    public void send(byte[] data) throws IOException {
            DatagramPacket dp = new DatagramPacket(buffer, buffer.length,
                InetAddress.getByName(remoteHost), sendPort);
            dp.setData(data);
            sendSocket.send(dp);
    }
    /**
     * 关闭UDP套接字
     */
    public void close() {
            try {
                    // 关闭接收套接字
                    if (receiveSocket.isConnected()) {
                            receiveSocket.disconnect();
                            receiveSocket.close();
```

更多资源请访问稀酷客(www.ckook.com)

```
                }
                // 关闭发送套接字
                if (sendSocket.isConnected()) {
                        sendSocket.disconnect();
                        sendSocket.close();
                }
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }
```

❑ 构建InitDialog

通过上述实现，我们可以构建一个简单的基于UDP协议的通信系统。在此基础上，我们需要通过一些简单的GUI界面配置相关参数。

我们将通过InitDialog类完成目标主机IP、本机IP、发送端口、接收端口和用户昵称5项信息的配置。有关Swing实现，不属于本书阐述内容，请读者参考相关的Java API。完整代码实现如代码清单12-17所示。

代码清单12-17 InitDialog

```
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.Frame;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.net.InetAddress;
import java.net.UnknownHostException;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
/**
 * @author 梁栋
 * @since 1.0
 */
public class InitDialog extends JDialog {
    private static final long serialVersionUID = -8482349275221329655L;
    // 默认宽度
    private static final int DEFAULT_WIDTH = 200;
    // 默认高度
    private static final int DEFAULT_HEIGHT = 210;
    // 接收端口
    private int receivePort;
    // 发送端口
```

更多资源请访问稀酷客(www.ckook.com)

```
private int sendPort;
// 用户昵称
private String username;
// 目标主机
private String remoteHost;
// 本地主机
private String localHost;
// 取消状态
private boolean cancelled = true;
// @return the localHost
public String getLocalHost() {
        return localHost;
}
// @return the cancelled
public boolean isCancelled() {
        return cancelled;
}
// @return the username
public String getUsername() {
        return username;
}
// @return the receivePort
public int getReceivePort() {
        return receivePort;
}
// @return the sendPort
public int getSendPort() {
        return sendPort;
}
// @return the remoteHost
public String getRemoteHost() {
        return remoteHost;
}
// @param owner
public InitDialog(Frame owner) {
        super(owner, "初始化对话框", true);
        // 初始化文本输入字段
        String local;
        try {
                local = InetAddress.getLocalHost().getHostAddress();
        } catch (UnknownHostException e) {
                local = "localhost";
        }
        final JTextField remoteHostField = new JTextField(local, 10);
        final JTextField localHostField = new JTextField(local, 10);
        final JTextField receivePortField = new JTextField("8001", 10);
        final JTextField sendPortField = new JTextField("8002", 10);
        final JTextField usernameField = new JTextField("zlex", 10);
```

更多资源请访问稀酷客(www.ckook.com)

```
// 构建输入面板
JPanel inputPanel = new JPanel();
inputPanel.setMinimumSize(new Dimension(80, 120));
inputPanel.setBorder(BorderFactory.createEtchedBorder());
inputPanel.add(new JLabel("目标主机: "));
inputPanel.add(remoteHostField);
inputPanel.add(new JLabel("本地主机: "));
inputPanel.add(localHostField);
inputPanel.add(new JLabel("接收端口: "));
inputPanel.add(receivePortField);
inputPanel.add(new JLabel("发送端口: "));
inputPanel.add(sendPortField);
inputPanel.add(new JLabel("用户昵称: "));
inputPanel.add(usernameField);
// 构建确认按钮
JButton okButton = new JButton("确定");
okButton.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            // 赋值
            remoteHost = remoteHostField.getText();
            localHost = localHostField.getText();
            receivePort = Integer.parseInt(receivePortField.getText());
            sendPort = Integer.parseInt(sendPortField.getText());
            username = usernameField.getText();
            cancelled = false;
            InitDialog.this.dispose();
        }
});
// 构建取消按钮
JButton cancelButton = new JButton("取消");
cancelButton.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            InitDialog.this.dispose();
        }
});
// 构建按钮面板
JPanel buttonPanel = new JPanel();
buttonPanel.add(okButton);
buttonPanel.add(cancelButton);
getContentPane().add(inputPanel, BorderLayout.CENTER);
getContentPane().add(buttonPanel, BorderLayout.SOUTH);
// 设置最小尺寸
setMinimumSize(new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT));
// 设置窗口大小不可调
setResizable(false);
// 窗口在屏幕中间显示
```

更多资源请访问稀酷客(www.ckook.com)

```
            setLocationRelativeTo(null);
            // 显示
            setVisible(true);
        }
    }
```

❑ 构建MainFrame

最后，我们将构建一个主体GUI窗口，用于初始化配置、消息的接收/发送等操作。Swing相关实现请读者参考Java API。完整代码如代码清单12-18所示。

代码清单12-18 MainFrame

```
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.io.IOException;
import java.net.SocketException;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JSplitPane;
import javax.swing.JTextArea;
import javax.swing.text.DefaultCaret;
/**
 * UDP协议聊天工具
 * @author 梁栋
 * @since 1.0
 */
public class MainFrame extends JFrame implements Runnable {
    private static final long serialVersionUID = 6479444953062332931;
    // 字符集
    private static final String CHARSET = "UTF-8";
    // UDP套接字
    private UDPSocket socket;
    // 初始化对话框
    private InitDialog initDialog;
    // 默认宽度
    public static final int DEFAULT_WIDTH = 500;
    // 默认高度
```

更多资源请访问稀酷客(www.ckook.com)

```
public static final int DEFAULT_HEIGHT = 400;
// 设置拆分窗格
private JSplitPane splitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT);
// 发送文本域
private JTextArea sendTextArea = new JTextArea();
// 接收文本域
private JTextArea receiveTextArea = new JTextArea();
// 按钮面板
private JPanel buttonPanel = new JPanel();
/**
 * 绑定主机和端口
 * @throws Exception
 */
public MainFrame() {
        // 优先启动初始化对话框
        this.initDialog = new InitDialog(this);
        // 若初始化对话框取消，则关闭系统，反之初始化系统
        if (initDialog.isCancelled()) {
                System.exit(0);
        } else {
                initSocket();
                initGUI();
        }
}
// 初始化套接字
public void initSocket() {
        try {
            socket = new UDPSocket(initDialog.getLocalHost(), initDialog.getRemoteHost
            (), initDialog.getReceivePort(), initDialog.getSendPort());
        } catch (SocketException e) {
            e.printStackTrace();
        }
}
// 初始化用户界面
public void initGUI() {
        setTitle("From: " + initDialog.getLocalHost() + " To: " + initDialog.
        getRemoteHost());
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setMinimumSize(new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT));
        getContentPane().add(splitPane, BorderLayout.CENTER);
        getContentPane().add(buttonPanel, BorderLayout.SOUTH);
        initReceivePanel();
        initSendPanel();
        initButtonPanel();
        // 窗口在屏幕中间显示
        setLocationRelativeTo(null);
        setVisible(true);
        setResizable(false);
```

更多资源请访问稀酷客(www.ckook.com)

```
}
// 初始化按钮面板
private void initButtonPanel() {
        JButton sendButton = new JButton("发送(S)");
        sendButton.setMnemonic(KeyEvent.VK_S);
        sendButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                try {
                    send(sendTextArea.getText());
                } catch (IOException ioe) {
                    ioe.printStackTrace();
                }
            }
        });
        JButton exitButton = new JButton("关闭(X)");
        exitButton.setMnemonic(KeyEvent.VK_X);
        exitButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                socket.close();
                System.exit(0);
            }
        });
        buttonPanel.setBorder(BorderFactory.createEtchedBorder());
        buttonPanel.add(sendButton);
        buttonPanel.add(exitButton);
}
// 初始化接收消息面板
private void initReceivePanel() {
        receiveTextArea.setEditable(false);
        JPanel panel = new JPanel();
        panel.setLayout(new BorderLayout());
        panel.setBorder(BorderFactory.createEtchedBorder());
        JLabel label = new JLabel();
        label.setText("接收到的消息: ");
        panel.add(label, BorderLayout.NORTH);
        JScrollPane scrollPane = new JScrollPane();
        scrollPane.getViewport().add(receiveTextArea);
        DefaultCaret caret = (DefaultCaret) receiveTextArea.getCaret();
        caret.setUpdatePolicy(DefaultCaret.ALWAYS_UPDATE);
        panel.add(scrollPane, BorderLayout.CENTER);
        panel.setMinimumSize(new Dimension(0, DEFAULT_WIDTH / 3));
        splitPane.add(panel);
}
// 初始化发送消息面板
private void initSendPanel() {
        JPanel panel = new JPanel();
```

更多资源请访问稀酷客(www.ckook.com)

```
panel.setLayout(new BorderLayout());
panel.setBorder(BorderFactory.createEtchedBorder());
JLabel label = new JLabel();
label.setText("待发送的消息：");
panel.add(label, BorderLayout.NORTH);
JScrollPane scrollPane = new JScrollPane();
scrollPane.setWheelScrollingEnabled(true);
DefaultCaret caret = (DefaultCaret) sendTextArea.getCaret();
caret.setUpdatePolicy(DefaultCaret.ALWAYS_UPDATE);
sendTextArea.addKeyListener(new KeyAdapter() {
    public void keyPressed(KeyEvent event) {
        // Ctrl+Enter组合键
        if ((event.getKeyCode() ==
        KeyEvent.VK_ENTER) && (event.isControlDown())) {
            try {
                send(sendTextArea.getText());
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
});
scrollPane.getViewport().add(sendTextArea);
panel.add(scrollPane, BorderLayout.CENTER);
        splitPane.add(panel);
}
/**
 * 接收消息
 * @throws IOException
 */
public void receive() throws IOException {
        String message = new String(socket.receive(), CHARSET);
        StringBuilder sb = new StringBuilder();
        sb.append(receiveTextArea.getText());
        sb.append(message);
        receiveTextArea.setText(sb.toString());
}
/**
 * 发送消息
 * @param message
 *消息
 * @throws IOException
 */
public void send(String message) throws IOException {
        if (message.isEmpty()) {
            return;
        }
        DateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        StringBuilder sendMessage = new StringBuilder();
```

更多资源请访问稀酷客(www.ckook.com)

```
            sendMessage.append(df.format(new Date()));
            sendMessage.append(" (" + initDialog.getUsername() + ") ");
            sendMessage.append("\r\n");
            sendMessage.append(message);
            sendMessage.append("\r\n");
            message = sendMessage.toString();
            socket.send(message.getBytes(CHARSET));
            StringBuilder receiveMessage = new StringBuilder(receiveTextArea.getText());
            receiveMessage.append(sendMessage);
            receiveTextArea.setText(receiveMessage.toString());
            sendTextArea.setText(null);
    }
    /*
     * (non-Javadoc)
     * @see java.lang.Runnable#run()
     */
    public void run() {
            // 循环读
            while (true) {
                try {
                    receive();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
    }
    /**
     * 总入口
     * @param args
     * @throws Exception
     */
    public static void main(String[] args) throws Exception {
            MainFrame mainFrame = new MainFrame();
            Thread t = new Thread(mainFrame);
            t.start();
    }
}
```

2. 验证服务

执行MainFrame类的main()方法，将获得初始化对话框，如图12-7所示。

这里，作者的主机IP为“192.168.1.2”，欲访问IP为“192.168.1.3”的主机。这里我们使用默认的端口号，将目标主机的值置为“192.168.1.3”，并指定用户昵称“snowolf”。

此外，我们将在IP为“192.168.1.3”的主机上执行MainFrame类的main()方法，进行相关配置，如图12-8所示。

这里，我们需要把目标主机指向IP“192.168.1.2”，将端口互换，并指定用户昵称“zlex”。

更多资源请访问稀酷客(www.ckook.com)

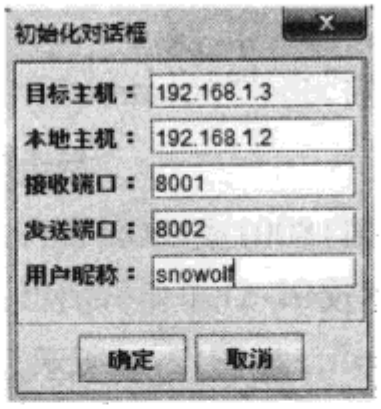


图12-7　初始化对话框1

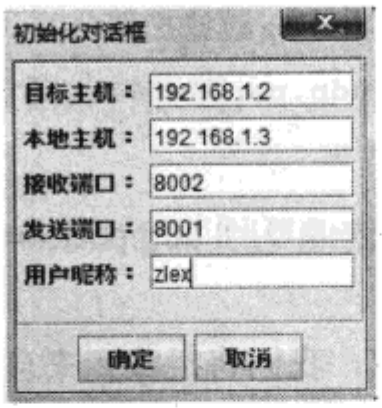


图12-8　初始化对话框2

此时，我们可以通过UDP协议进行聊天了，如图12-9所示。“snowolf”和“zlex”互相问候。

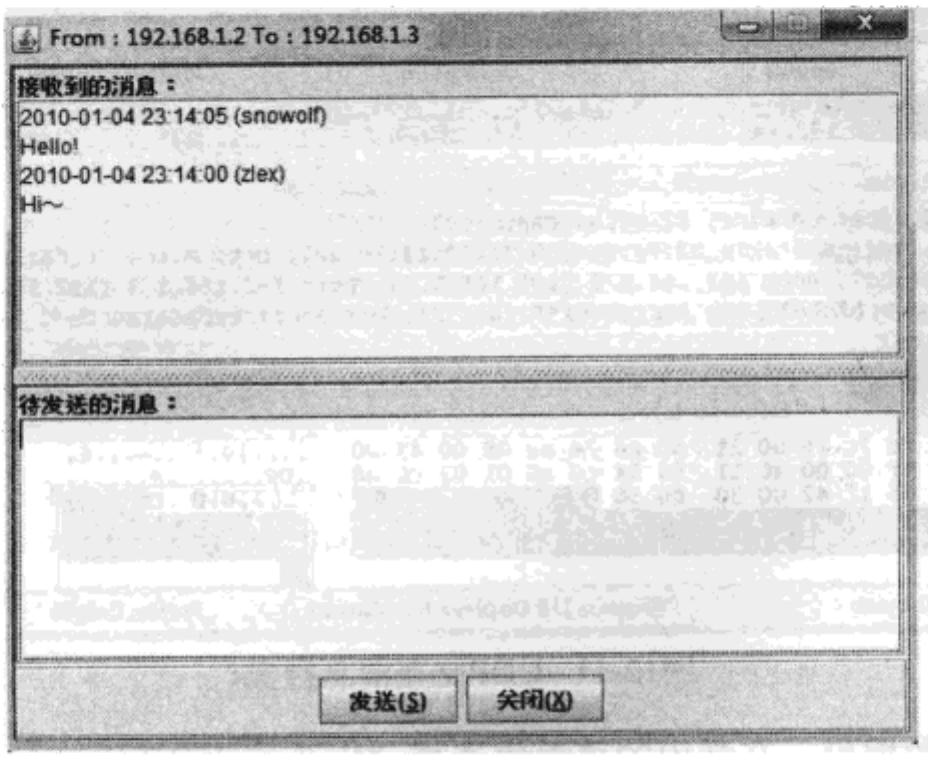


图12-9　UDP聊天窗口2

3. 网络监测

在这里，作者将使用网络监测工具Wireshark监测UDP交互数据。作者将Wireshark绑定在本机（IP为192.168.1.2）上，请读者根据实际情况绑定对应网卡。网卡绑定如图12-10所示。

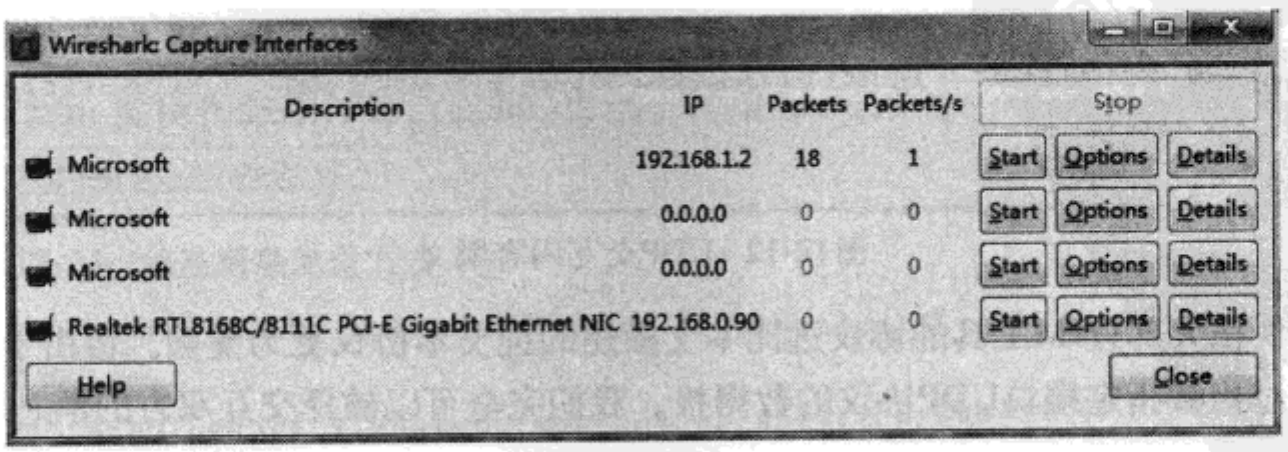


图12-10　网卡绑定

完成网卡绑定后，在过滤器地址栏中输入过滤信息，如代码清单12-19所示。

更多资源请访问稀酷客(www.ckook.com)

代码清单12-19 过滤拦截1

```
udp && (udp.port==8001 || udp.port == 8002)
```

其中

(udp.port==8001 || udp.port == 8002)　　限定UDP端口8001或8002

udp　　指定UDP协议

执行拦截，我们将拦截到来自“snowolf”的问候：“Hello”，如图12-11所示。

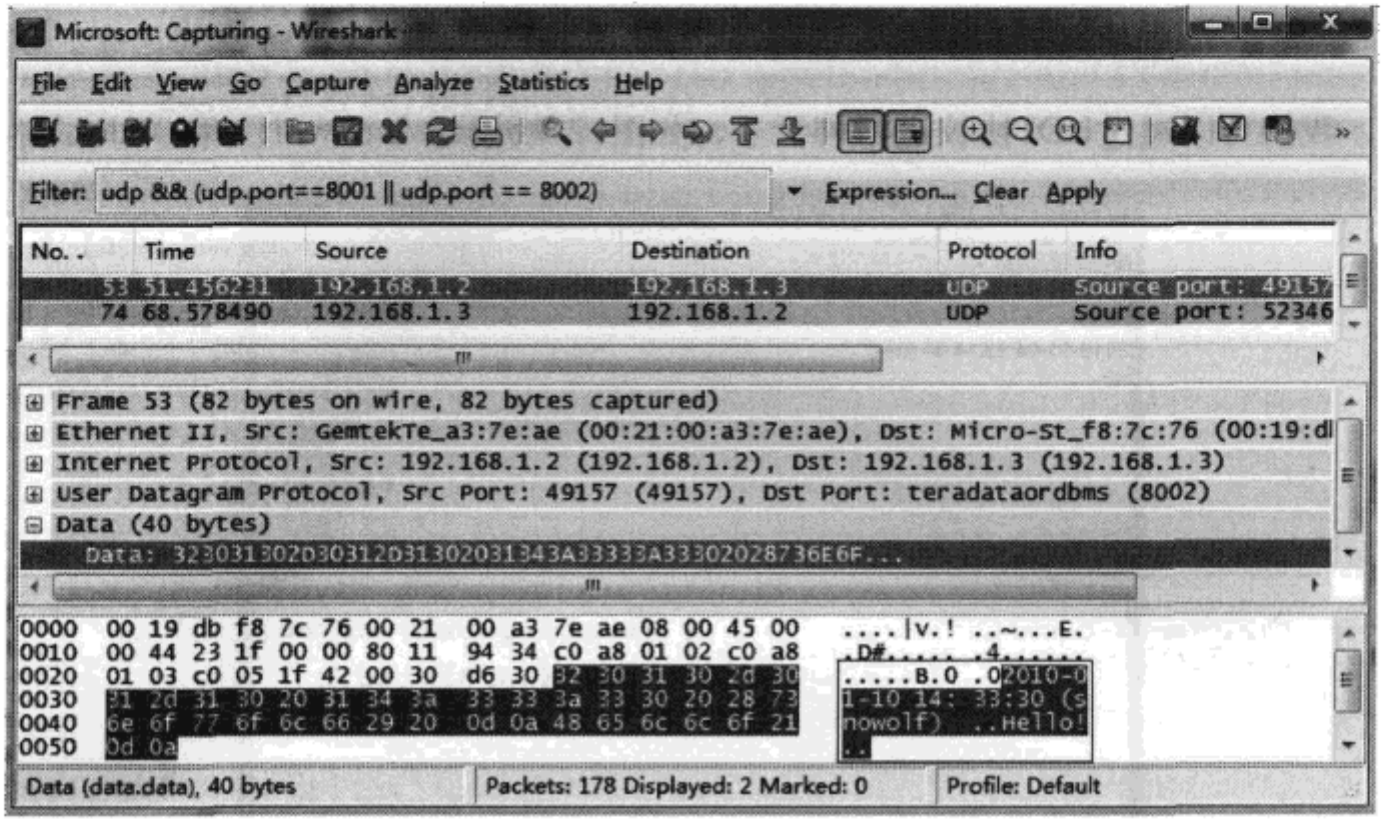


图12-11 UDP交互消息拦截1

右键单击第一条数据包，在弹出的菜单中选择“Follow UDP Stream”，我们将得到此次聊天的内容，如图12-12所示。

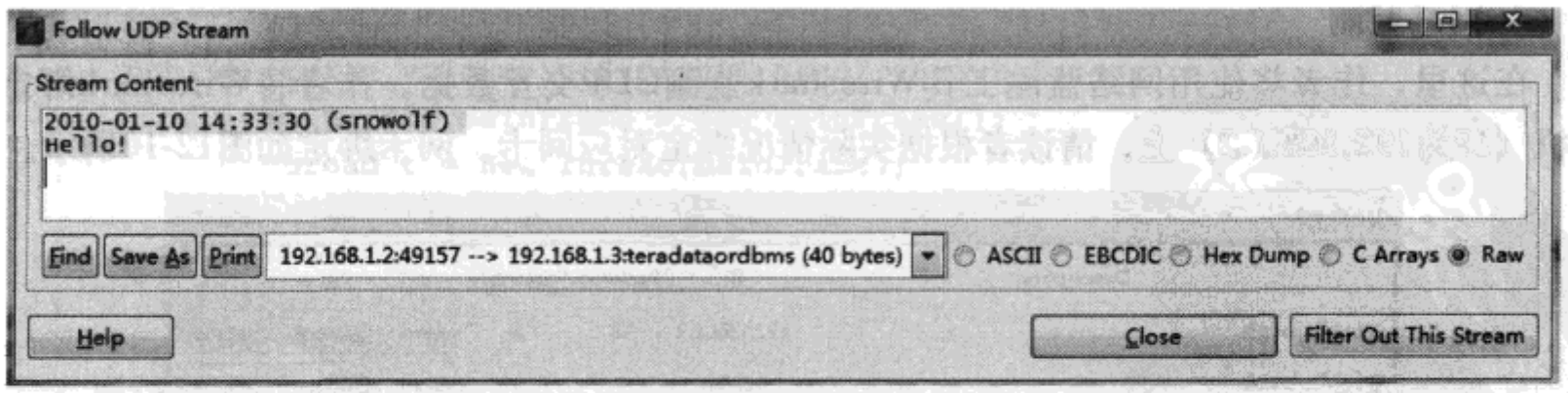


图12-12 UDP交互内容明文

当然，现在大部分IM工具的协议远比本文描述的纯文本协议更为复杂。但由于协议本身必须公开，通过拦截指定端口UDP协议的数据报，我们完全可以破译交互双方的消息内容。

12.2.2 安全升级1——隐藏数据

Base64算法并不属于加密算法领域，但却常常用于数据隐藏。当然，这样做并不能起到数据加密的作用，但同样起到了隐藏数据的作用。我们可以对UDPChat稍作修改，隐藏用户交互

更多资源请访问稀酷客(www.ckook.com)

的数据。

本书将使用Commons Codec提供的Base64算法实现类对数据进行隐藏。

1. 增强安全性

这里，我们将构建Security类，并提供encrypt()和decrypt()方法分别用于加密和解密操作。完整代码实现如代码清单12-20所示。

代码清单12-20　Security类1

```
import org.apache.commons.codec.binary.Base64;
/**
 * 安全组件
 * @author 梁栋
 * @since 1.0
 */
public abstract class Security {
    /**
     * 加密
     * @param data
     *              待加密数据
     * @return byte[] 加密数据
     */
    public static byte[] encrypt(byte[] data) {
        return Base64.encodeBase64(data);
    }
    /**
     * 解密
     * @param data
     *              待解密数据
     * @return byte[] 解密数据
     */
    public static byte[] decrypt(byte[] data) {
        return Base64.decodeBase64(data);
    }
}
```

对MainFrame类稍作修改，修改send()方法，使用Security类的encrypt()方法对数据编码，如代码清单12-21所示。

代码清单12-21　修改消息发送方法

```
public void send(String message) throws IOException {
    // 省略
    socket.send(Security.encrypt(message.getBytes(CHARSET)));
    // 省略
}
```

修改receive ()方法，使用Security类的decrypt()方法对数据解码，如代码清单12-22所示。

更多资源请访问稀酷客(www.ckook.com)

代码清单12-22 修改消息接收方法

```
public void receive() throws IOException {
    // 解密
    byte[] data = Security.decrypt(socket.receive());
    String message = new String(data, CHARSET);
    // 省略
}
```

接下来，我们将监测通过Base64算法增强安全性后的UDPChat。

2. 网络监测

我们重复前面的操作，使用UDPChat进行交互，并通过Wireshark对UDPChat交互数据包监测，如图12-13所示。

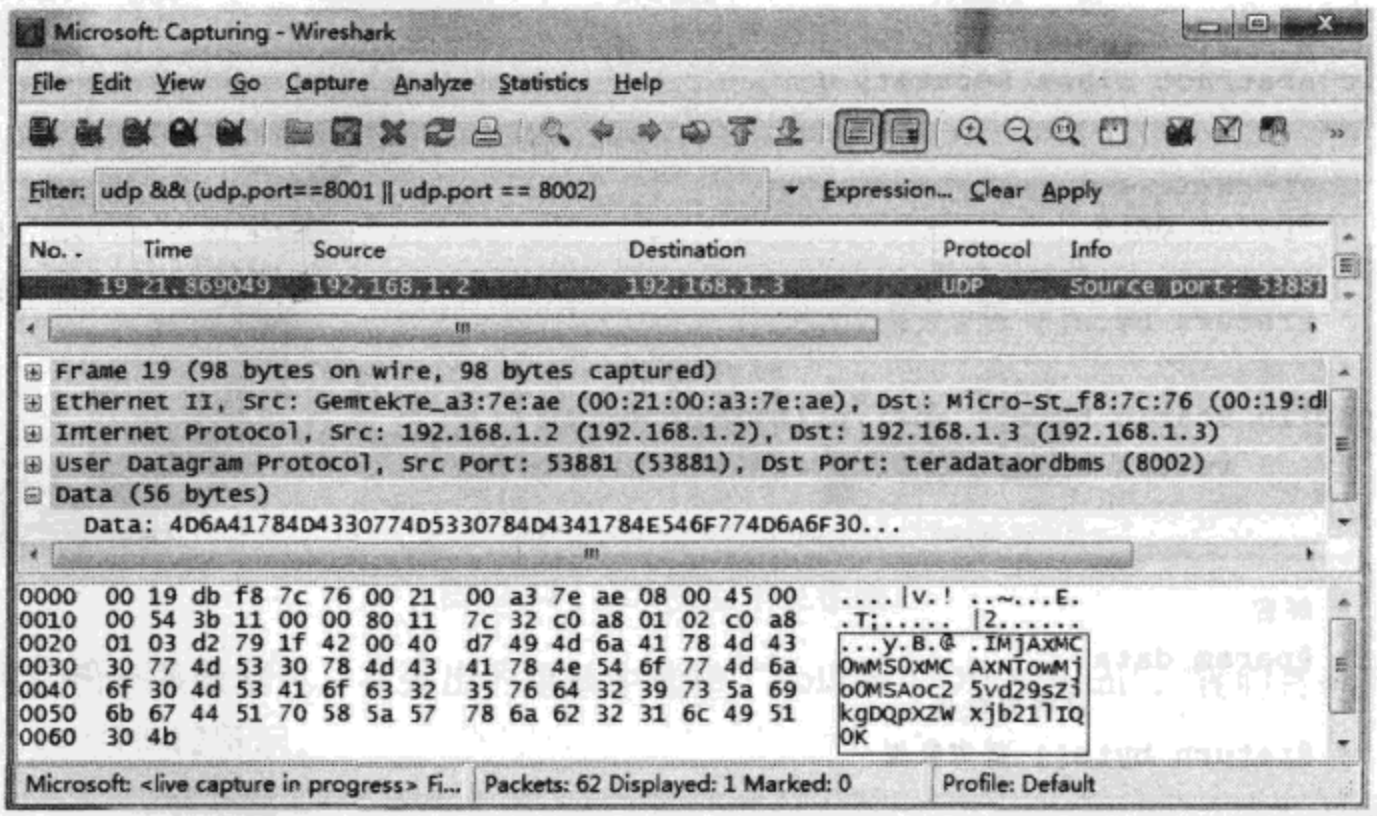


图12-13 UDP交互消息拦截2

打开“Follow UDP Stream”窗口，我们得到的将是一串Base64编码字符串，如图12-14所示。

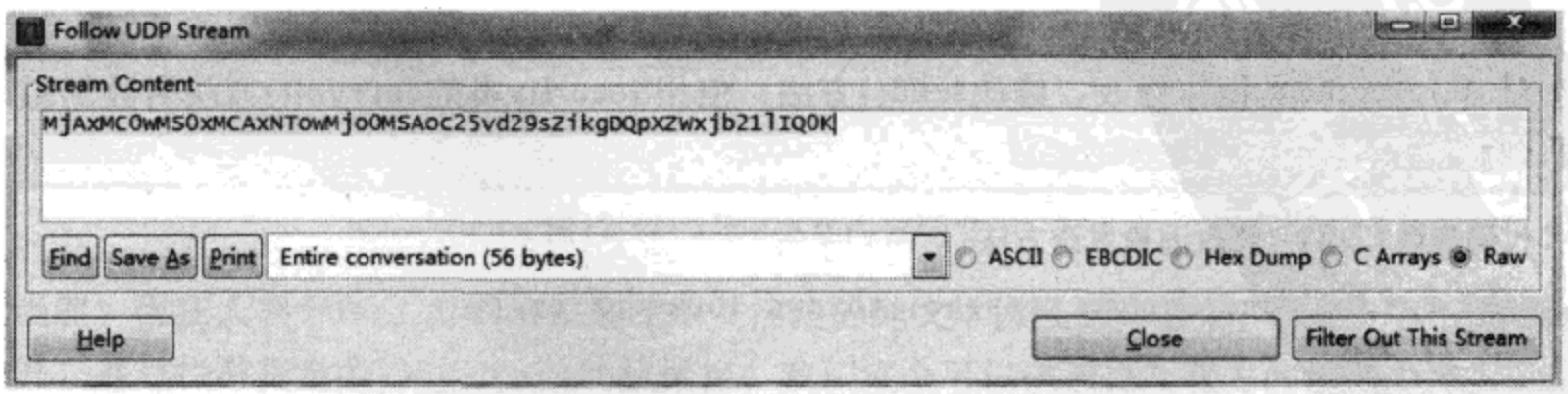


图12-14 UDP交互内容Base64编码

实际上，我们发送了一条问候消息，如图12-15所示。

通过第5章的阐述，我们知道经过Base64编码的数据本身并不安全，任何监听者都可以对其进行破解。

更多资源请访问稀酷客(www.ckook.com)

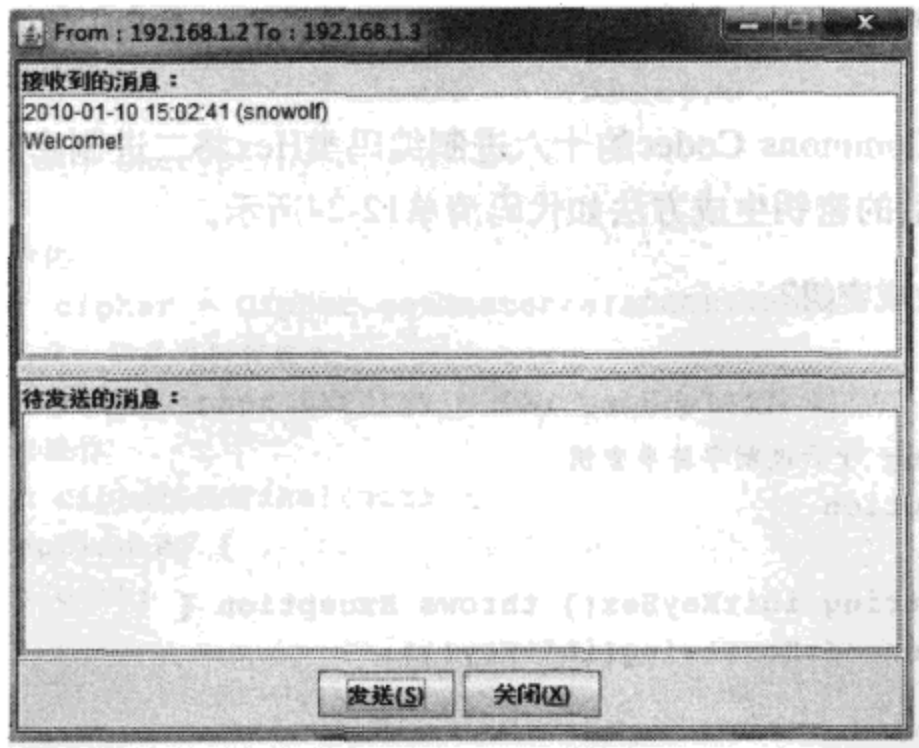


图12-15　UDP聊天窗口Base64编码

虽然，Base64编码数据并不安全，但在实际应用中很多软件都采用Base64编码的方式对数据进行隐藏。想要真正起到保护数据机密性的作用，需采用对称加密算法或非对称加密算法对数据加密。

12.2.3　安全升级2——加密数据

想要真正加强数据的机密性，对称加密算法是我们的首选。在实际应用中，类似于这样的网络交互系统几乎都采用了对称加密算法对数据进行加密。这些系统通常用于传递聊天消息，甚至是传递对账数据。

本文将使用AES算法对UDPChat交互数据进行加密/解密。

1. 增强安全性

我们继续对Security类进行修改，使用AES算法对数据加密。

首先，我们需要构建密钥，如代码清单12-23所示。

代码清单12-23　生成密钥1

```
/**
 * 生成密钥
 * @return byte[] 二进制密钥
 * @throws Exception
 */
public static byte[] initKey() throws Exception {
    // 实例化
    KeyGenerator kg = KeyGenerator.getInstance(ALGORITHM);
    kg.init(256);
    // 生成秘密密钥
    SecretKey secretKey = kg.generateKey();
    // 获得密钥的二进制编码形式
```

更多资源请访问稀酷客(www.ckook.com)

```
        return secretKey.getEncoded();
    }
```

这里，我们使用Commons Codec的十六进制编码类Hex将二进制密钥转换为十六进制字符串，使其可见，改进后的密钥生成方法如代码清单12-24所示。

代码清单12-24　生成密钥2

```
/**
 * 生成密钥
 * @return String 十六进制字符串密钥
 * @throws Exception
 */
public static String initKeyHex() throws Exception {
    return Hex.encodeHexString(initKey());
}
```

我们将获得密钥（1486c5dc751a54ce3a58701ba537ecc8e257bf66127837e9401acdaceb6023f8），并将该密钥绑定在变量Key中。

反之，我们将十六进制密钥转换获得最终的秘密密钥，如代码清单12-25所示。

代码清单12-25　转换密钥

```
// 省略
/**
 * 密钥
 */
private static final String KEY = "1486c5dc751a54ce3a58701ba
537ecc8e257bf66127837e9401acdaceb6023f8";
// 省略
/**
 * 转换密钥
 * @throws Exception
 */
private static Key getKey() throws Exception {
    byte[] key = Hex.decodeHex(KEY.toCharArray());
    // 实例化AES密钥材料
    SecretKey secretKey = new SecretKeySpec(key, ALGORITHM);
    return secretKey;
}
```

在不改变原Security类方法的前提下，我们仍使用encrypt()方法对数据加密，使用decrypt()方法对数据解密。

修改后的加密方法实现如代码清单12-26所示。

代码清单12-26　加密方法1

```
/**
 * 加密
 * @param data 待加密数据
```

更多资源请访问稀酷客(www.ckook.com)

```
 * @return byte[] 加密数据
 * @throws Exception
 */
public static byte[] encrypt(byte[] data) {
    try {
            // 实例化
            Cipher cipher = Cipher.getInstance(ALGORITHM);
            // 初始化，设置为加密模式
            cipher.init(Cipher.ENCRYPT_MODE, getKey());
            // 执行操作
            return cipher.doFinal(data);
    } catch (Exception e) {
            return data;
    }
}
```

修改后的解密方法实现如代码清单12-27所示。

代码清单12-27 解密方法1

```
/**
 * 解密
 * @param data 待解密数据
 * @return byte[] 解密数据
 * @throws Exception
 */
public static byte[] decrypt(byte[] data) {
    try {
            // 实例化
            Cipher cipher = Cipher.getInstance(ALGORITHM);
            // 初始化，设置为解密模式
            cipher.init(Cipher.DECRYPT_MODE, getKey());
            // 执行操作
            return cipher.doFinal(data);
    } catch (Exception e) {
            return data;
    }
}
```

完整实现如代码清单12-28所示。

代码清单12-28 Security类2

```
import java.security.Key;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import org.apache.commons.codec.binary.Hex;
/**
```

更多资源请访问稀酷客(www.ckook.com)

```
 * 安全组件
 * @author 梁栋
 * @since 1.0
 */
public abstract class Security {
    // 密钥算法
    public static final String ALGORITHM = "AES";
    // 密钥
    private static final String KEY =
    "1486c5dc751a54ce3a58701ba537ecc8e257bf66127837e9401acdaceb6023f8";
    /**
     * 转换密钥
     * @throws Exception
     */
    private static Key getKey() throws Exception {
            byte[] key = Hex.decodeHex(KEY.toCharArray());
            // 实例化AES密钥材料
            SecretKey secretKey = new SecretKeySpec(key, ALGORITHM);
            return secretKey;
    }
    /**
     * 解密
     * @param data 待解密数据
     * @return byte[] 解密数据
     * @throws Exception
     */
    public static byte[] decrypt(byte[] data) {
            try {
                    // 实例化
                    Cipher cipher = Cipher.getInstance(ALGORITHM);
                    // 初始化，设置为解密模式
                    cipher.init(Cipher.DECRYPT_MODE, getKey());
                    // 执行操作
                    return cipher.doFinal(data);
            } catch (Exception e) {
                    return data;
            }
    }
    /**
     * 加密
     * @param data 待加密数据
     * @return byte[] 加密数据
     * @throws Exception
     */
    public static byte[] encrypt(byte[] data) {
            try {
                    // 实例化
                    Cipher cipher = Cipher.getInstance(ALGORITHM);
```

更多资源请访问稀酷客(www.ckook.com)

```
            // 初始化，设置为加密模式
            cipher.init(Cipher.ENCRYPT_MODE, getKey());
            // 执行操作
            return cipher.doFinal(data);
        } catch (Exception e) {
            return data;
        }
}
/**
 * 生成密钥
 * @return byte[] 二进制密钥
 * @throws Exception
 */
public static byte[] initKey() throws Exception {
        // 实例化
        KeyGenerator kg = KeyGenerator.getInstance(ALGORITHM);
        kg.init(256);
        // 生成秘密密钥
        SecretKey secretKey = kg.generateKey();
        // 获得密钥的二进制编码形式
        return secretKey.getEncoded();
}
/**
 * 生成密钥
 * @return String 十六进制字符串密钥
 * @throws Exception
 */
public static String initKeyHex() throws Exception {
        return Hex.encodeHexString(initKey());
}
}
```

2. 网络监测

有了加密算法的庇护，我们可以高枕无忧了，即便是将我们的银行卡号发送给对方，也不必担心，如图12-16所示。

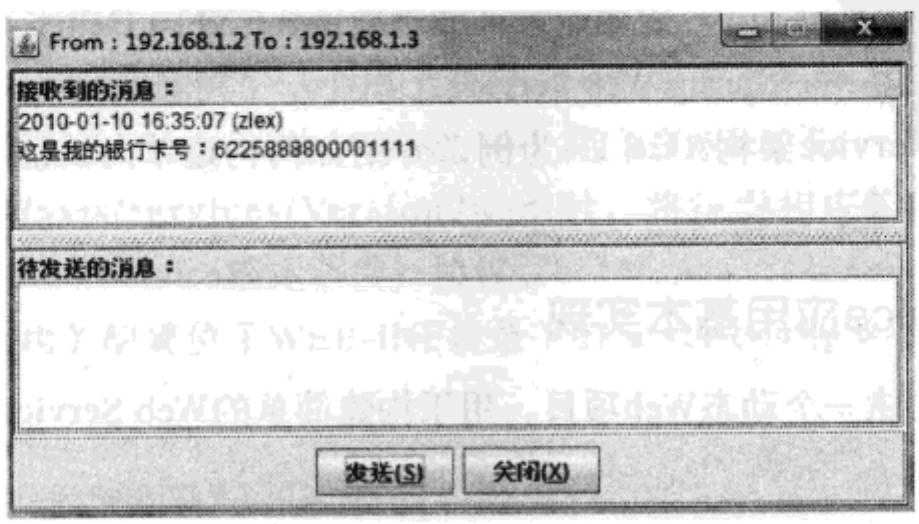


图12-16　UDP聊天窗口AES算法加密

更多资源请访问稀酷客(www.ckook.com)

截获此次交互的数据包，如图12-17所示。

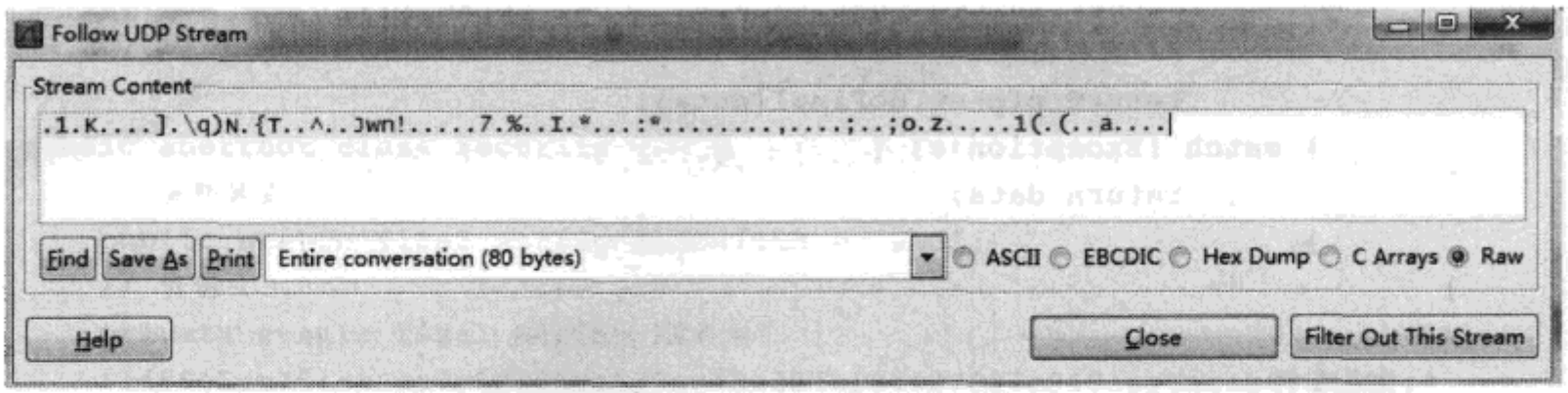


图12-17 UDP交互内容AES算法加密

如果没有密钥，我们将无法在有效的时间内破译加密内容，也就失去了破译的意义。

如果需要对交互的数据进行验证，我们可以使用消息摘要算法对消息内容做摘要处理，并对解密后的数据做摘要验证。相关内容请读者参考第6章自行实现。

12.3 实例：Web Service应用开发安全

相信大多数读者对于开发Web Service系统并不陌生。通过Web Service我们可以获得天气预报、火车时刻表、股票行情、电视节目预告等相应数据。公司之间的合作也常常通过Web Service系统完成数据同步等。

如何保护这样一个基于明文协议（WSDL、SOAP协议等）的系统是每一个架构师不容忽视的安全问题。

如果对传输的对象加密，或者对传输数据做加密处理，虽然可以保证在接口开放的前提下保护敏感数据，但却加大了安全开发的工作量，同时也将带来许多新问题：密钥的管理、数据交互双方算法的商榷、敏感数据范围的定义等。

使用数字证书构建SSL/TLS协议，最终构建HTTPS服务是保护Web Service系统的最佳方案。

开源组织Apache为我们提供了非常丰富的Web Service框架（http://ws.apache.org/），其中尤以Axis和CXF最为常用。Axis框架包含两个分支：Axis（http://ws.apache.org/axis/）和Axis2（http://ws.apache.org/axis2/）。CXF框架（http://cxf.apache.org/）由Celtix（http://celtix.ow2.org/）和XFire（http://xfire.codehaus.org/）发展而来，准确地说，CXF = Celtix + XFire。Spring也提供了自己的Web Service产品——Spring Web Service。

本文以开源Web Service架构Axis 1.4为例，介绍如何构建单向认证服务和双向认证服务，确保Web Service开发安全。

12.3.1 Web Service应用基本实现

我们在Eclipse中构建一个动态Web项目，用于构建简单的Web Service应用。

1. 准备工作

根据Axis官方文档说明，我们将用到以下jar包：

更多资源请访问稀酷客(www.ckook.com)

- axis.jar：Axis核心包。
- activation.jar 和mail.jar：用于电子邮件操作。
- jaxrpc.jar：用于基于XML的远程过程调用。
- commons-discovery.jar：用最佳的算法查找某个接口的所有已知的实现。
- commons-logging.jar：用于日志控制。
- wsdl4j.jar：生成/解析WSDL协议。

请读者朋友在Axis官方网站下载完整Axis发行包，上述jar包均可以在该发行包中找到。

接下来，我们需要配置web.xml文件，完整内容如代码清单12-29所示。

代码清单12-29　web.xml文件

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
id="WebApp_ID"
    version="2.5">
    <display-name>Apache-Axis</display-name>
    <servlet>
        <display-name>Apache-Axis Servlet</display-name>
        <servlet-name>axis</servlet-name>
        <servlet-class>org.apache.axis.transport.http.AxisServlet</servlet-class>
        <load-on-startup>0</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>axis</servlet-name>
        <url-pattern>/services/*</url-pattern>
    </servlet-mapping>
</web-app>
```

org.apache.axis.transport.http.AxisServlet类是我们唯一需要配置的Servlet。这里我们将请求路径位于“/services/”下的任何请求都交给AxisServlet类处理。

org.apache.axis.Version类提供了一个用于获得Axis版本信息的静态方法getVersion()。我们可以将该方法暴露在Web Service中，构建一个简单的Web Service。当我们访问地址http://localhost:8080/axis/services/Version?wsdl时，将得到相应的WSDL（Web Service Definition Language，Web Service描述语言）协议。

Axis Web Service相关配置位于WEB-INF目录下的server-config.wsdd文件中。文件详情如代码清单12-30所示。

代码清单12-30　server-config.wsdd文件

```
<?xml version="1.0" encoding="UTF-8"?>
```

更多资源请访问稀酷客(www.ckook.com)

```
<deployment
    xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
    <transport
        name="http">
        <requestFlow>
            <handler
                type="java:org.apache.axis.handlers.http.URLMapper" />
        </requestFlow>
    </transport>
    <transport
        name="local">
        <responseFlow>
            <handler
                type="java:org.apache.axis.transport.local.LocalResponder" />
        </responseFlow>
    </transport>
    <service
        name="Version"
        provider="java:RPC">
        <parameter
            name="allowedMethods"
            value="getVersion" />
        <parameter
            name="className"
            value="org.apache.axis.Version" />
    </service>
</deployment>
```

注意上述配置中的“service”节点，这里我们将暴露一个名为“Version”的服务。该服务指向org.apache.axis.Version类，仅允许使用getVersion()方法完成相应操作。

现在，我们启动Tomcat并访问地址http://localhost:8080/axis/services，将得到Web Service列表，如图12-18所示。



图12-18 Web Service列表1

继续点击上述页面链接（wsdl）获得Version对应WSDL协议，如图12-19所示。

通过上述页面的描述，我们已经很清楚：可以通过调用getVersion()方法获得String类型的返回值。

更多资源请访问稀酷客(www.ckook.com)

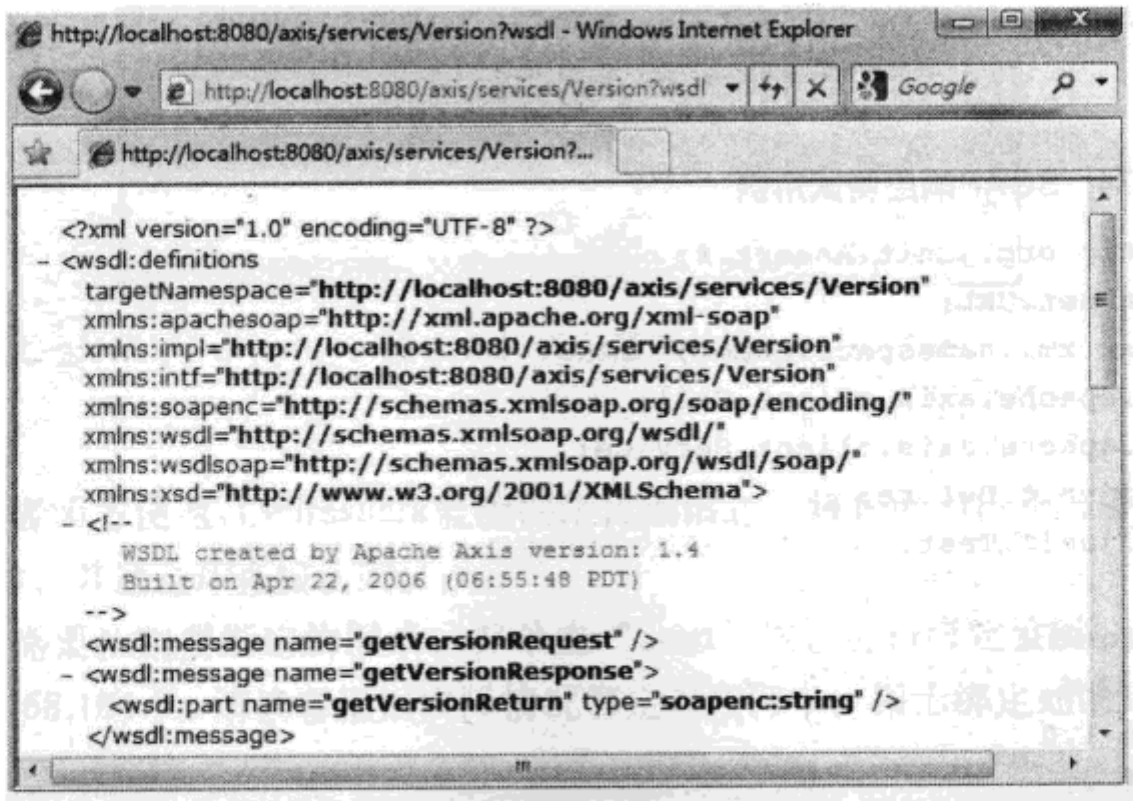


图12-19 WSDL协议1

2. 验证服务

现在，我们构建相应的SOAP（Simple Object Access Protocol，简单对象访问协议）响应测试用例，验证上述服务。

Axis框架提供的Web Service客户端访问需要先构建Service实现类，并通过该类的createCall()方法创建调用对象Call类实例化对象。完整代码如代码清单12-31所示。

代码清单12-31 构建调用对象

```
// 创建调用对象
Service service = new Service();
Call call = (Call) service.createCall();
```

接下来，我们需要对Call类实例化对象call做相应设置：配置远程调用的方法（通过调用setOperationName()方法）和设置访问URL（通过调用setTargetEndpointAddress()方法）。完整代码如代码清单12-32所示。

代码清单12-32 配置远程调用对象

```
// 调用远程方法
call.setOperationName(new QName(namespaceUri, "getVersion"));
// 设置URL
call.setTargetEndpointAddress(new URL(wsdlUrl));
```

最后，我们需要通过invoke()方法完成调用，并获得返回值。完整代码如代码清单12-33所示。

代码清单12-33 执行远程调用

```
// 执行远程调用，同时获得返回值
String version = (String) call.invoke(new Object[] {});
```

更多资源请访问稀酷客(www.ckook.com)

有关Axis框架的相关实现，请读者查阅相关API文档。测试用例完整代码如代码清单12-34所示。

代码清单12-34 SOAP响应测试用例

```
import static org.junit.Assert.*;
import java.net.URL;
import javax.xml.namespace.QName;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.junit.Before;
import org.junit.Test;
/**
 * WebService测试
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class WebServiceTest {
    // Namespace URL
    private String namespaceUri = "http:// localhost:8080/axis/services/Version";
    // WSDL URL
    private String wsdlUrl = "http://localhost:8080/axis/services/Version?wsdl";
    /**
     * 测试
     * @throws Exception
     */
    @Test
    public final void test() throws Exception {
        // 创建调用对象
        Service service = new Service();
        Call call = (Call) service.createCall();
        // 调用远程方法
        call.setOperationName(new QName(namespaceUri, "getVersion"));
        // 设置URL
        call.setTargetEndpointAddress(new URL(wsdlUrl));
        // 执行远程调用，同时获得返回值
        String version = (String) call.invoke(new Object[] {});
        // 打印信息
        System.err.println(version);
        // 验证
        assertNotNull(version);
    }
}
```

如果调用验证通过，我们将在控制台得到Axis的版本信息，如图12-20所示。

更多资源请访问稀酷客(www.ckook.com)

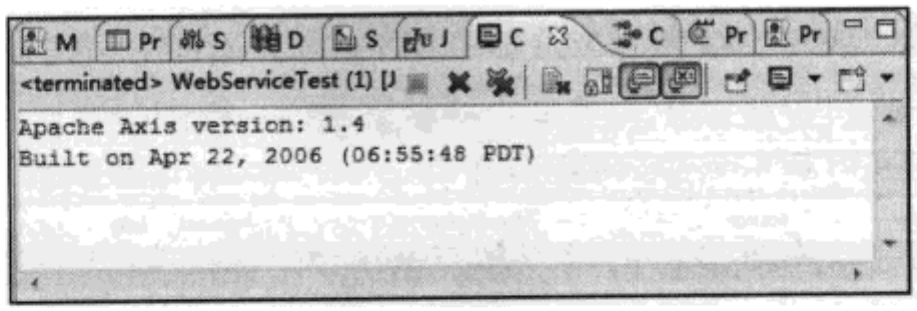


图12-20　Axis版本信息

3. 网络监测

在这里，作者为方便通过Wireshark监测网络传输信息，将Tomcat应用部署在虚拟机（IP为192.168.184.131），并通过IP直接访问。

首先，我们需要找到要绑定的网卡，并单击“Start”按钮进行绑定监听。作者使用的虚拟机网卡IP为192.168.184.1，请读者根据实际情况绑定对应网卡。网卡绑定如图12-21所示。

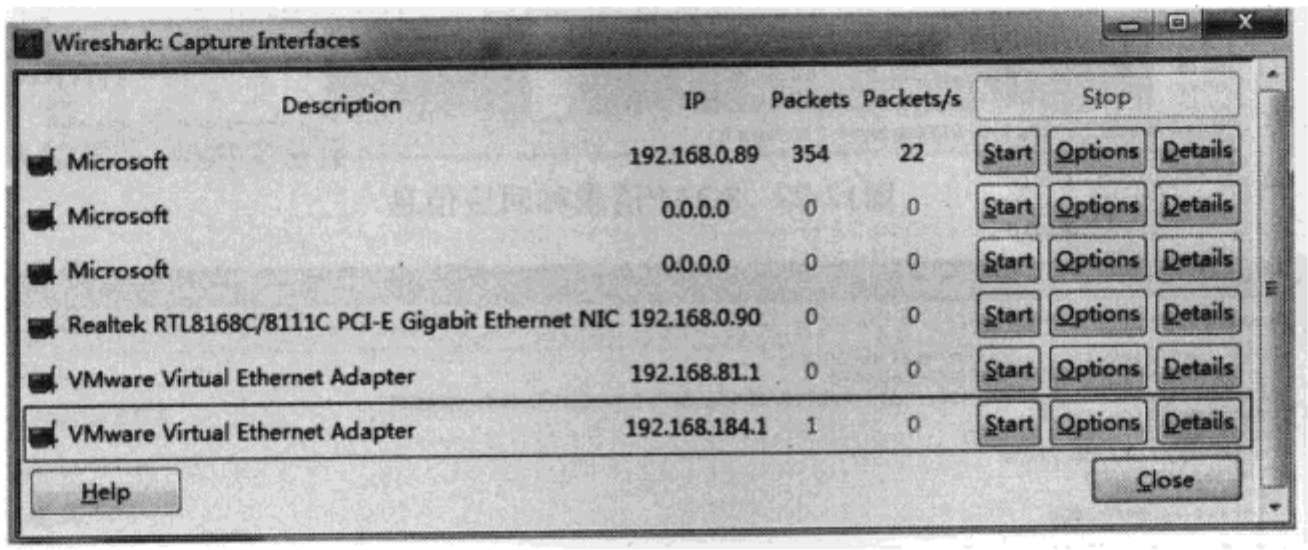


图12-21　网卡绑定

完成网卡绑定后，在过滤器地址栏中输入过滤信息，如代码清单12-35所示。

代码清单12-35　过滤拦截1

```
(ip.src==192.168.184.131 && ip.dst==192.168.184.1) || (ip.dst==192.168.184.131
&& ip.src==192.168.184.1) && http
```

其中

(ip.src==192.168.184.131 && ip.dst==192.168.184.1)　指限定由本机请求虚拟机

(ip.dst==192.168.184.131 && ip.src==192.168.184.1)　指限定由虚拟机回复本机

http　指定HTTP协议

重新执行SOAP请求，我们将在Wireshark中获得详细的SOAP请求和回应信息，如图12-22所示。

右键点击图12-22中任意一条数据包（No.298或No.300），在弹出的菜单中选择“Follow TCP Stream”菜单项。

在弹出的窗口（Follow TCP Stream）中单击下拉列表框，选择请求和回复信息，分别得到SOAP请求和SOAP回复内容，如图12-23和图12-24所示。

更多资源请访问稀酷客(www.ckook.com)

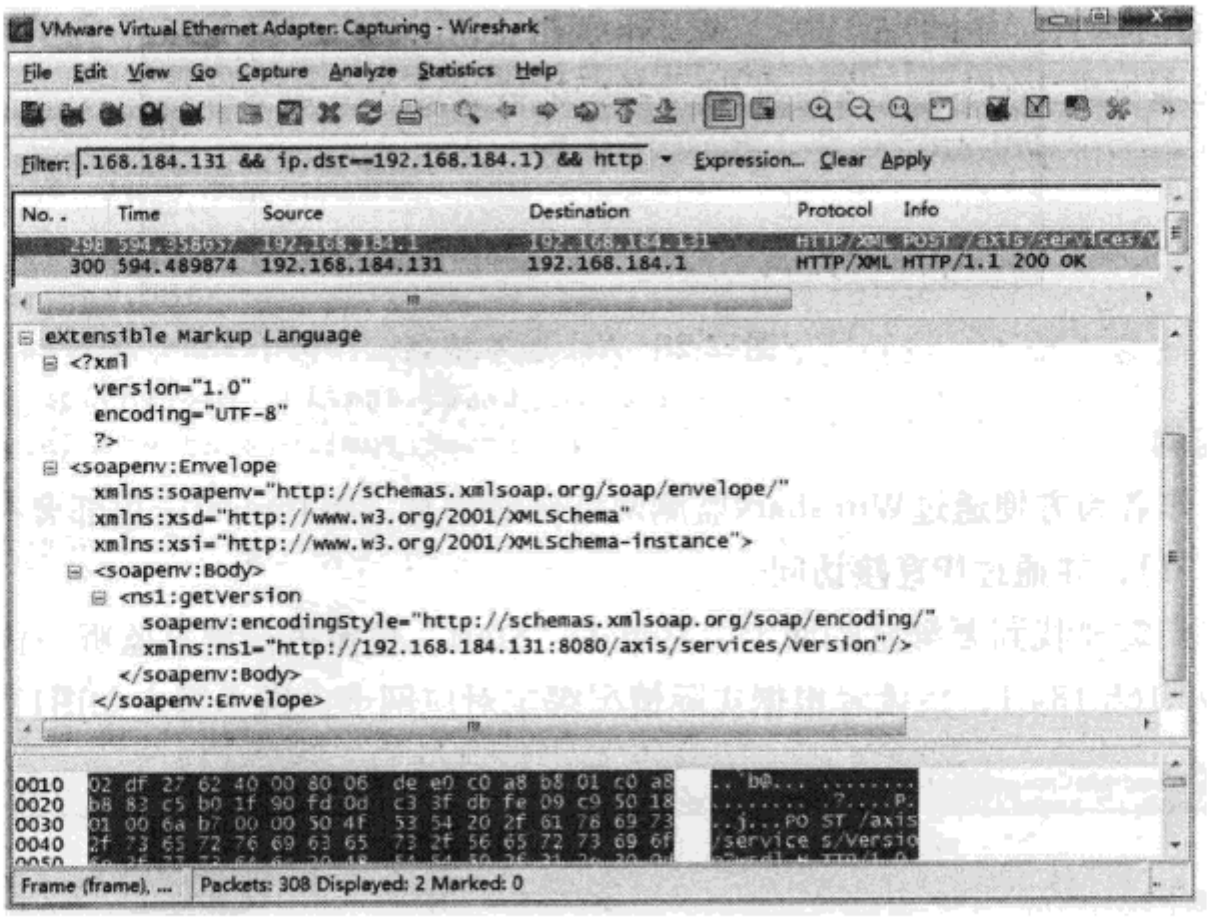


图12-22 SOAP请求和回应信息

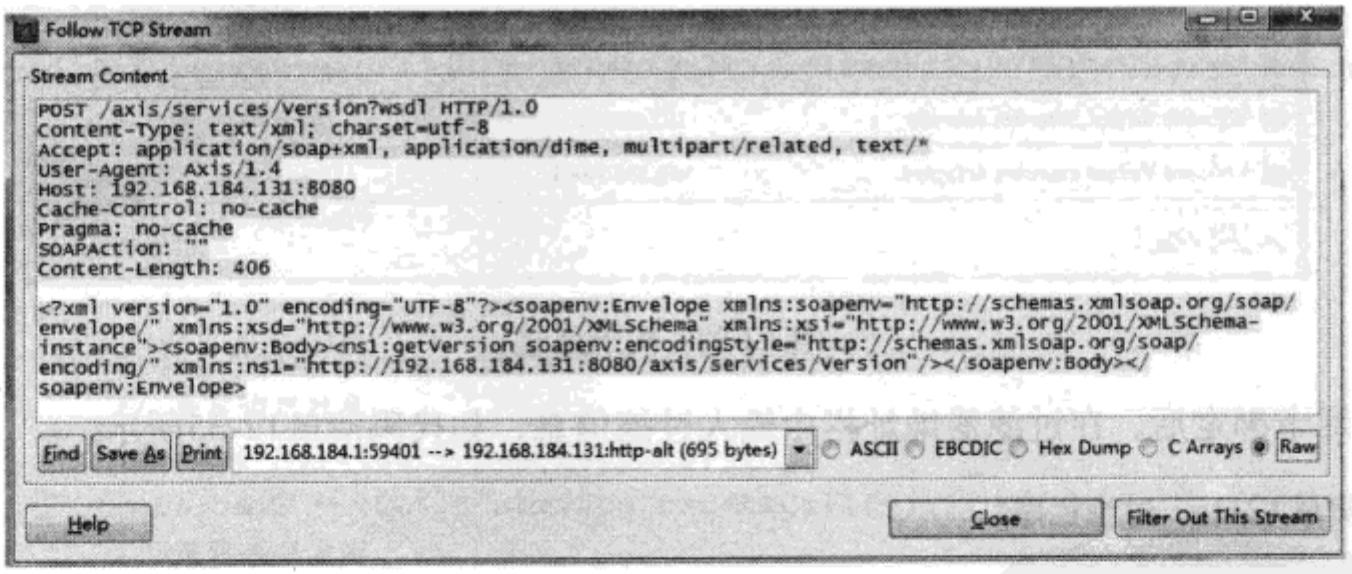


图12-23 SOAP请求

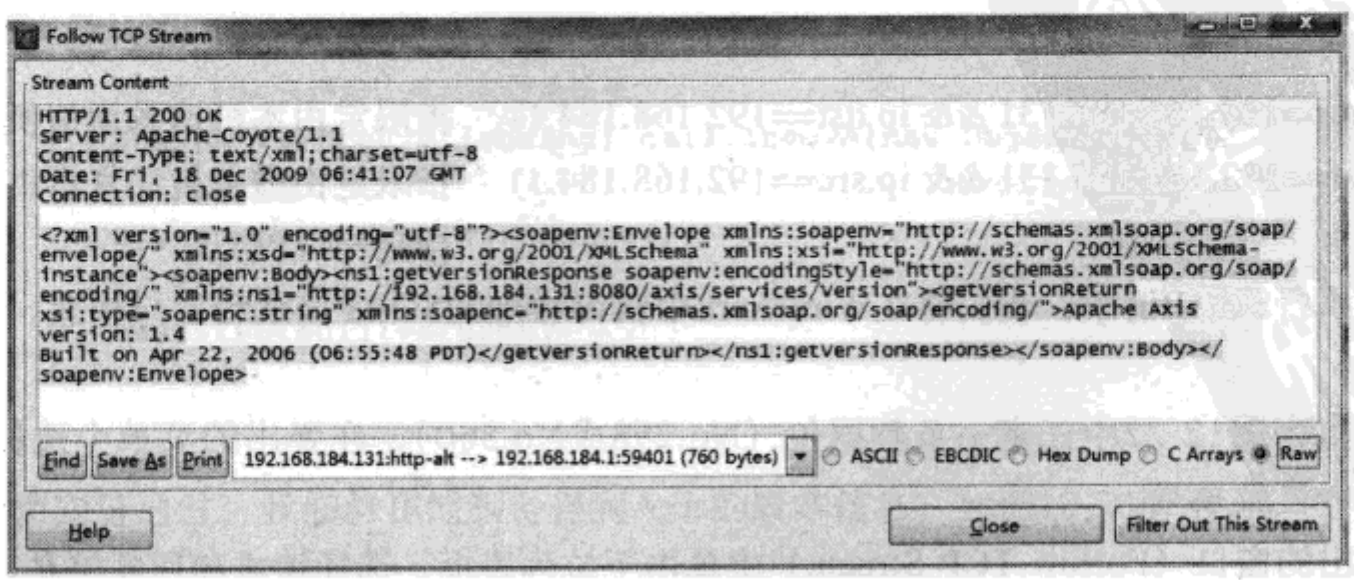


图12-24 SOAP回复

更多资源请访问稀酷客(www.ckook.com)

如果我们仅仅将列车时刻表、天气预报和电视节目预报等公开服务通过Web Service系统暴露服务，将无须考虑任何安全问题。但如果我们需要通过Web Service系统与合作伙伴交互商业数据，这将是一件可怕的事情。

12.3.2 安全升级1——单向认证服务

为构建单向认证服务，我们需要构建相应的数字证书，配置Tomcat服务器，并修改测试用例。最后，我们将通过Wireshark监测网络中真正传输的SOAP内容是否加密成功。

1. 构建证书

为构建单向认证服务，我们需要构建相应的数字证书。请读者参考第10章，完成OpenSSL相关配置并构建CA根证书目录。

❑ 初始化设置

在命令行下，将当前路径切换至CA根证书目录执行命令完成初始化设置。完整命令如代码清单12-36所示。

代码清单12-36　初始化设置

```
echo off
echo 构建目录
mkdir certs
mkdir crl
mkdir newcerts
mkdir private
echo 构建文件
echo 0>index.txt
echo 01>serial
echo 构建随机数 private/.rand
openssl rand -out private/.rand 1000
```

❑ 构建CA根证书

接下来，我们需要构建CA根证书。完整命令如代码清单12-37所示。

代码清单12-37　构建CA根证书

```
echo 构建根证书私钥 private/ca.key.pem
openssl genrsa -aes256 -out private/ca.key.pem 2048
echo 生成根证书请求 private/ca.csr
openssl req -new -key private/ca.key.pem -out private/ca.csr -subj
"/C=CN/ST=BJ/L=BJ/O=zlex/OU=zlex/CN=Zlex CA"
echo 签发根证书 private/ca.cer
openssl x509 -req -days 10000 -sha1 -extensions v3_ca -signkey
private/ca.key.pem -in private/ca.csr -out certs/ca.cer
echo 根证书转换 private/ca.p12
openssl pkcs12 -export -clcerts -in certs/ca.cer -inkey private/ca.key.pem -out
certs/ca.p12
```

更多资源请访问稀酷客(www.ckook.com)

上述命令执行过程中需要输入相应密码，此处一律输入密码“123456”。

❑ 导入证书

为方便演示，需将自签名CA个人信息交换文件ca.p12导入IE浏览器。点击IE浏览器“工具”菜单，在弹出菜单中点击“Internet选项”。在弹出的“Internet选项”对话框中选择“内容”选项卡，在证书栏中点击“证书”按钮。在弹出的“证书”对话框中选择“受信任的根证书颁发机构”选项卡，点击“导入”按钮导入CA个人信息交换文件ca.p12，在弹出的对话框中输入CA根证书密码（这里为“123456”），并确认导入该证书。

最终，我们将在“受信任的根证书颁发机构”选项卡中找到我们导入的自签名CA根证书，如图12-25所示。

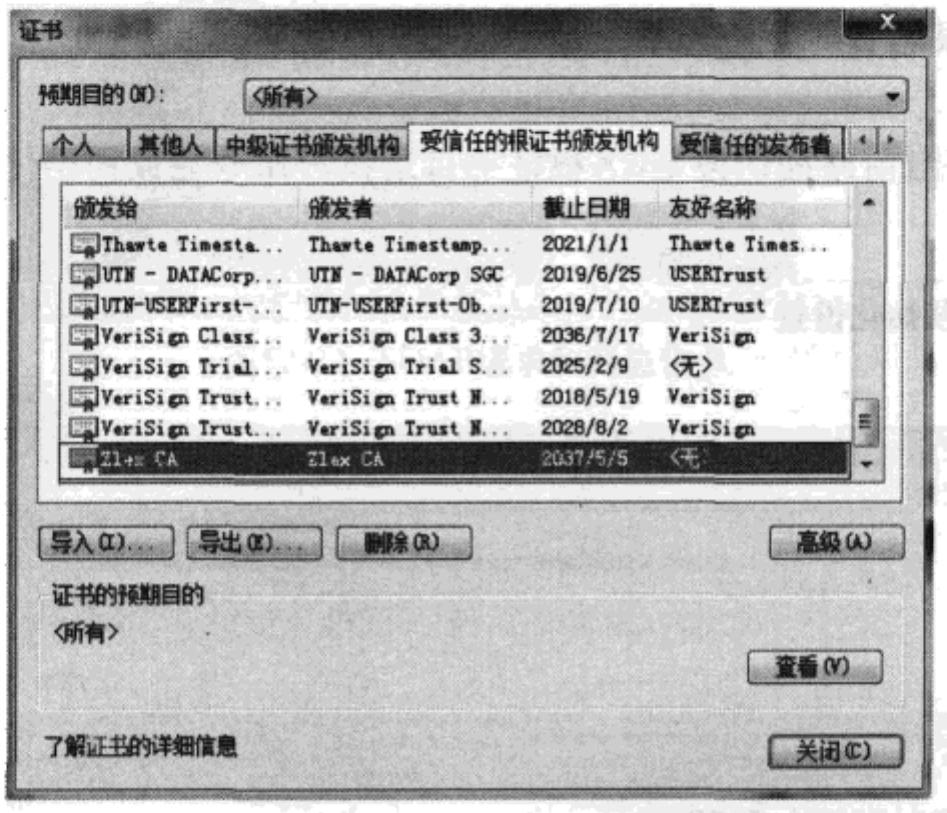


图12-25 导入的自签名CA根证书

❑ 构建服务器证书

接下来，我们需要构建服务器证书。这里，作者将应用部署在虚拟机（IP：192.168.184.131）上，故使用IP作为用户名称构建数字证书，读者可根据实际情况指定相应的IP地址或域名构建服务器证书。完整命令如代码清单12-38所示。

代码清单12-38 构建服务器证书

```
echo 构建服务器证书私钥 private/server.key.pem
openssl genrsa -aes256 -out private/server.key.pem 2048
echo 生成服务器证书请求 private/server.csr
openssl req -new -key private/server.key.pem -out private/server.csr -subj
"/C=CN/ST=BJ/L=BJ/O=zlex/OU=zlex/CN=192.168.184.131"
echo 签发服务器证书 private/server.cer
openssl x509 -req -days 3650 -sha1 -extensions v3_req -CA certs/ca.cer -CAkey
private/ca.key.pem -CAserial ca.srl -CAcreateserial -in private/server.csr -out
certs/server.cer
```

更多资源请访问稀酷客(www.ckook.com)

```
echo 服务器证书转换 private/server.p12
openssl pkcs12 -export -clcerts -in certs/server.cer -inkey private/server.key.pem
-out certs/server.p12
```

上述命令执行过程中需要输入相应密码，此处为方便演示，一律输入密码“123456”。

❑ 导出服务器证书私钥

为了能通过Wireshark监测网络中加密的数据，我们需要将服务器密钥以明文形式导出。完整命令如代码清单12-39所示。

代码清单12-39　导出服务器证书私钥

```
echo 导出服务器证书私钥（明文） private/serkey.pem
openssl pkcs12 -in certs/server.p12 -out private/serkey.pem -nodes -nocerts
```

其中

pkcs12	PKCS#12编码格式证书命令
-in	表示输入文件，这里为certs/server.p12
-out	表示输出文件，这里为private/serkey.pem
-nodes	不加密（No DES）
-nocerts	不导出证书

执行上述命令时，请输入密码“123456”，执行结果如图12-26所示。

```
导出服务器证书私钥（明文） private/serkey.pem
Enter Import Password:
MAC verified OK
```

图12-26　导出服务器密钥

接下来，我们将使用ca.p12和server.p12个人信息交换文件构建单向认证服务。

2. 配置Tomcat

配置单向认证服务，需将上述操作获得的个人信息交换文件（ca.p12和server.p12文件）复制到Tomcat的conf目录下作为密钥库文件，并配置server.xml文件。server.xml文件详细配置如代码清单12-40所示。

代码清单12-40　server.xml配置——单向认证

```
<Connector
    clientAuth="false"
    SSLEnabled="true"
    maxThreads="150"
    port="443"
    protocol="HTTP/1.1"
    scheme="https"
    secure="true"
    sslProtocol="TLS"
    keystoreFile="conf/server.p12"
    keystorePass="123456"
```

更多资源请访问稀酷客(www.ckook.com)

```
keystoreType="PKCS12"
truststoreFile="conf/ca.p12"
truststorePass="123456"
truststoreType="PKCS12" />
```

注意，参数clientAuth值为“false”，即不需要客户端认证——单向认证。这里我们将端口改为443（port="443"），直接通过HTTPS访问。

同时，将密钥库文件指向server.p12文件（keystoreFile="conf/server.p12"），并指定密码为“123456”（keystorePass="123456"），同时标明密钥库文件类型为PKCS#12（keystoreType = "PKCS12"）。

此外，我们需要配置信任库相关属性。将信任库文件指向ca.p12文件（truststoreFile = "conf/ca.p12"），并指定密码为“123456”（truststorePass ="123456"），同时标明信任库文件类型为PKCS#12（truststoreType ="PKCS12"）。

为方便访问HTTPS服务，这里将端口指向HTTPS服务默认端口443，即将端口参数port值置为“443”。

重新启动Tomcat，我们通过IP直接访问地址https://192.168.184.131/axis/services，查看服务列表，如图12-27所示。

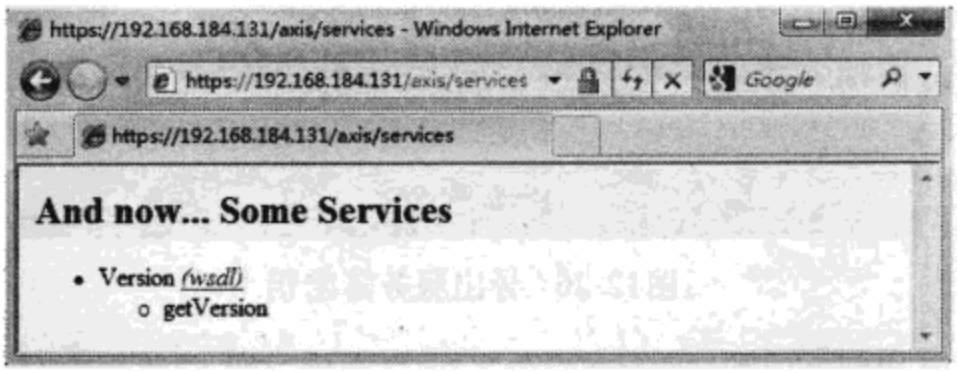


图12-27 Web Service列表2

如果未能导入CA根证书，将提示“证书错误”，如图12-28所示。

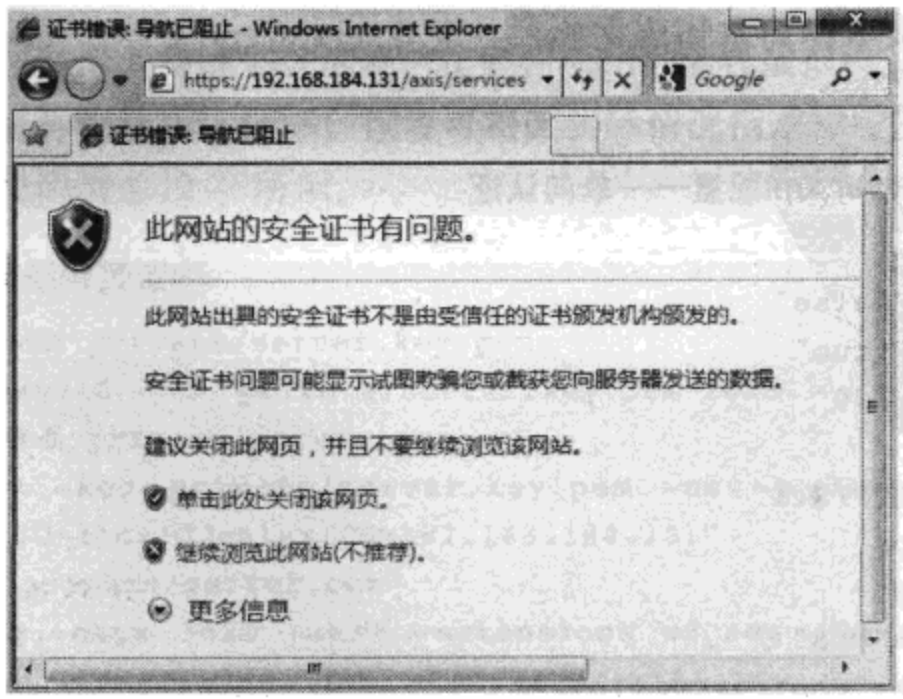


图12-28 证书错误

更多资源请访问稀酷客(www.ckook.com)

此时，请导入CA个人信息交换文件ca.p12，并重新打开IE浏览器访问该地址。

接下来，我们直接通过IE浏览器访问地址https://192.168.184.131/axis/services/Version?wsdl，获得WSDL协议，如图12-29所示。

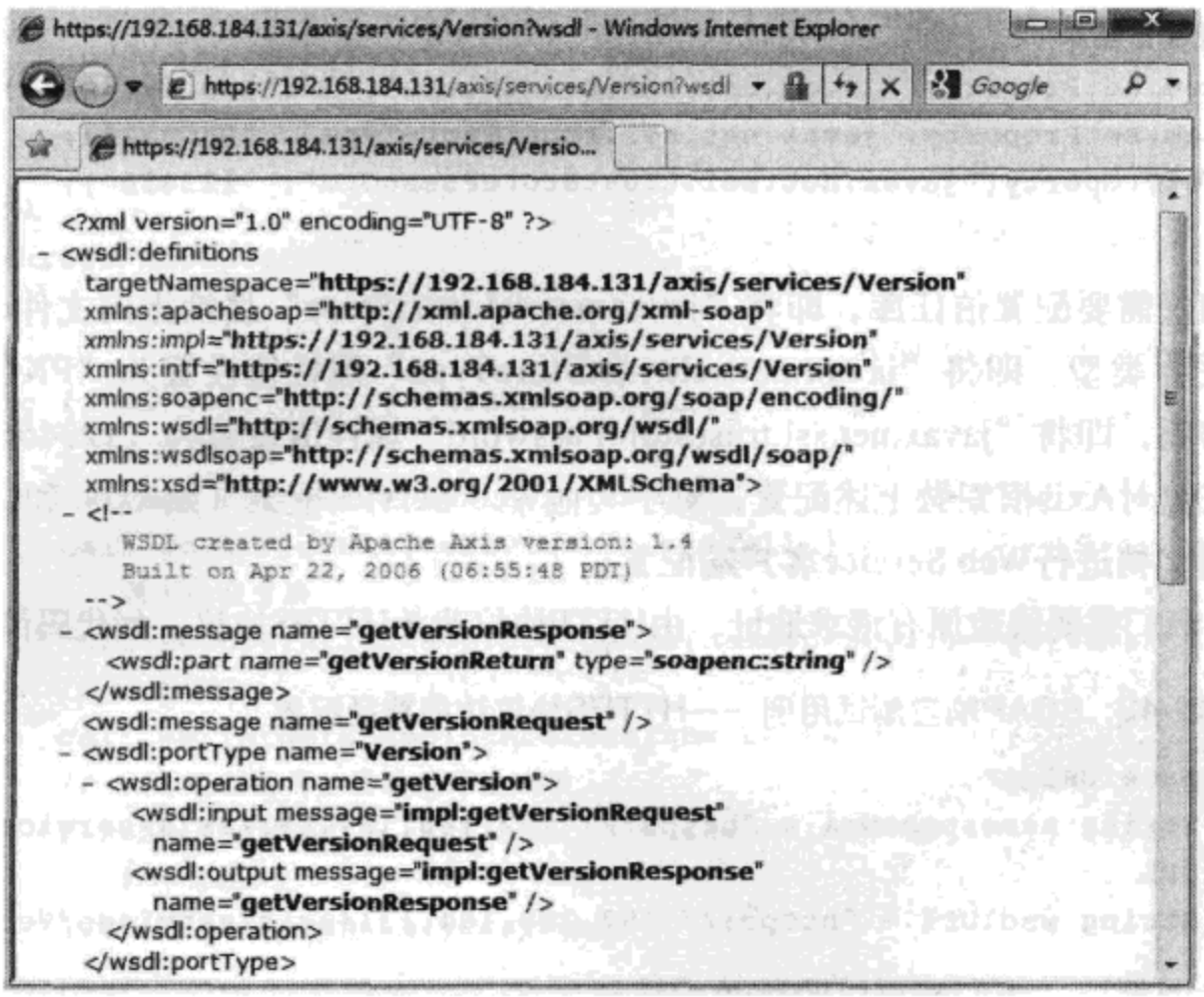


图12-29　WSDL协议2

此时，WSDL协议已经被加密。仔细对比图12-19和图12-29，我们会发现，原有指向Version服务的协议已经发生变化——由HTTP协议变为HTTPS协议。稍后我们将使用Wireshark监测网络数据的加密情况。

3. 验证服务

针对Axis架构，我们需要在进行SOAP交互之前初始化证书配置。在原有测试用例代码的基础上通过JUnit前置动作（注解@Before标注的方法，此处为init()方法）初始化信任库配置，即调用System类的setProperty()方法配置信任库的相关内容。

信任库的相关属性详细描述如下所示：

- javax.net.ssl.trustStore：指向信任库文件路径。
- javax.net.ssl.trustStoreType：信任库文件类型。
- javax.net.ssl.trustStorePassword：信任库密码。

对于密钥库相关配置也可按照上述方式如法炮制，我们将在后面介绍双向认证服务时详述。

此时，我们需要将服务器个人信息交换文件server.p12放置在D盘根目录下。初始化证书配置方法实现如代码清单12-41所示。

更多资源请访问稀酷客(www.ckook.com)

代码清单12-41 初始化证书配置——单向认证

```
// 初始化证书配置
@Before
public final void init() {
    // 配置信任库
    System.setProperty("javax.net.ssl.trustStore", "D:\\server.p12");
    System.setProperty("javax.net.ssl.trustStoreType", "PKCS12");
System.setProperty("javax.net.ssl.trustStorePassword", "123456");
}
```

此处我们仅需要配置信任库，即将“javax.net.ssl.trustStore”属性指向文件client.p12，并指定信任库文件类型，即将“javax.net.ssl.trustStoreType”属性值设置为“PKCS12”。同时，指定信任库密码，即将“javax.net.ssl.trustStorePassword”属性值设置为“123456”。

这里仅仅针对Axis框架做上述配置，对于其他Web Service框架（如Axis2和CXF框架），请读者根据相关文档进行Web Service客户端配置。

接下来，我们需要修改原有请求地址，由HTTP协议改为HTTPS协议，如代码清单12-42所示。

代码清单12-42 SOAP响应测试用例——HTTPS协议访问路径配置

```
// Namespace URL
private String namespaceUri = "https:// 192.168.184.131/axis/services/Version";
// WSDL URL
private String wsdlUrl = "https:// 192.168.184.131/axis/services/Version?wsdl";
```

完成上述调整后，我们就可以验证单向认证服务了。完整测试用例如代码清单12-43所示。

代码清单12-43 SOAP响应测试用例——单向认证

```
import static org.junit.Assert.*;
import java.net.URL;
import javax.xml.namespace.QName;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.junit.Before;
import org.junit.Test;
/**
 * WebService测试
 * @author 梁栋
 * @version 1.0
 * @since 1.0
 */
public class WebServiceTest {
    // Namespace URL
    private String namespaceUri = "https:// 192.168.184.131/axis/services/Version";
    // WSDL URL
    private String wsdlUrl = "https:// 192.168.184.131/axis/services/Version?wsdl";
    // 初始化证书配置
    @Before
```

更多资源请访问稀酷客(www.ckook.com)

```
public final void init() {
      // 配置信任库
       System.setProperty("javax.net.ssl.trustStore",
               "D:\\server.p12");
       System.setProperty("javax.net.ssl.trustStoreType", "PKCS12");
System.setProperty("javax.net.ssl.trustStorePassword", "123456");
}
/**
 * 测试
 * @throws Exception
 */
@Test
public final void test() throws Exception {
       // 创建调用对象
       Service service = new Service();
       Call call = (Call) service.createCall();
       // 调用远程方法
       call.setOperationName(new QName(namespaceUri, "getVersion"));
       // 设置URL
       call.setTargetEndpointAddress(new URL(wsdlUrl));
       // 执行远程调用，同时获得返回值
       String version = (String) call.invoke(new Object[] {});
       // 打印信息
       System.err.println(version);
              // 验证
              assertNotNull(version);
     }
}
```

执行上述测试用例时，init()方法将在执行test()方法之前被调用，完成初始化设置。

此时，我们可以通过加密方式进行SOAP请求和回复操作。稍后我们将通过Wireshark监测并解析加密SOAP数据。

4. 网络监测

打开Wireshark并绑定网卡（作者在这里绑定虚拟机网卡，IP为192.168.184.1）。在Wireshark过滤器地址栏中输入过滤信息，如代码清单12-44所示。

代码清单12-44　过滤拦截2

```
(ip.src==192.168.184.131 && ip.dst==192.168.184.1) || (ip.dst==192.168.184.131
&& ip.src==192.168.184.1)
```

其中

(ip.src==192.168.184.131 && ip.dst==192.168.184.1)　　指限定由本机请求虚拟机

(ip.dst==192.168.184.131 && ip.src==192.168.184.1)　　指限定由虚拟机回复本机

重新执行SOAP响应测试用例（单向认证），Wireshark监测结果如图12-30所示。

更多资源请访问稀酷客(www.ckook.com)

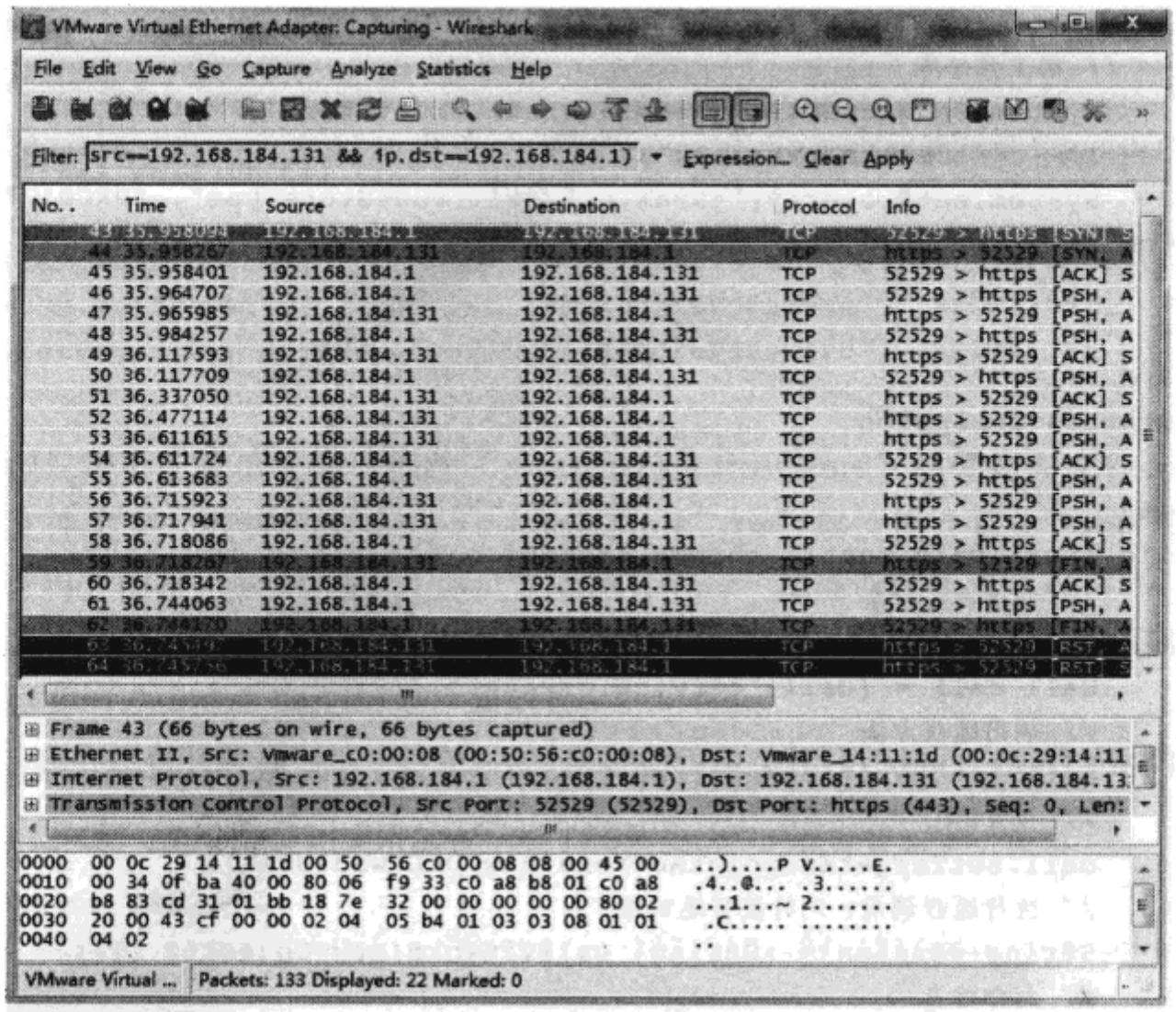


图12-30 Wireshark监测结果1

此时，我们无法在Wireshark中监测到真正的SSL/TLS交互内容。右键单击上图中任意一条数据包，在弹出的菜单中选择“Follow TCP Stream”菜单项。我们获得的将是完全无法识别的内容，如图12-31所示。

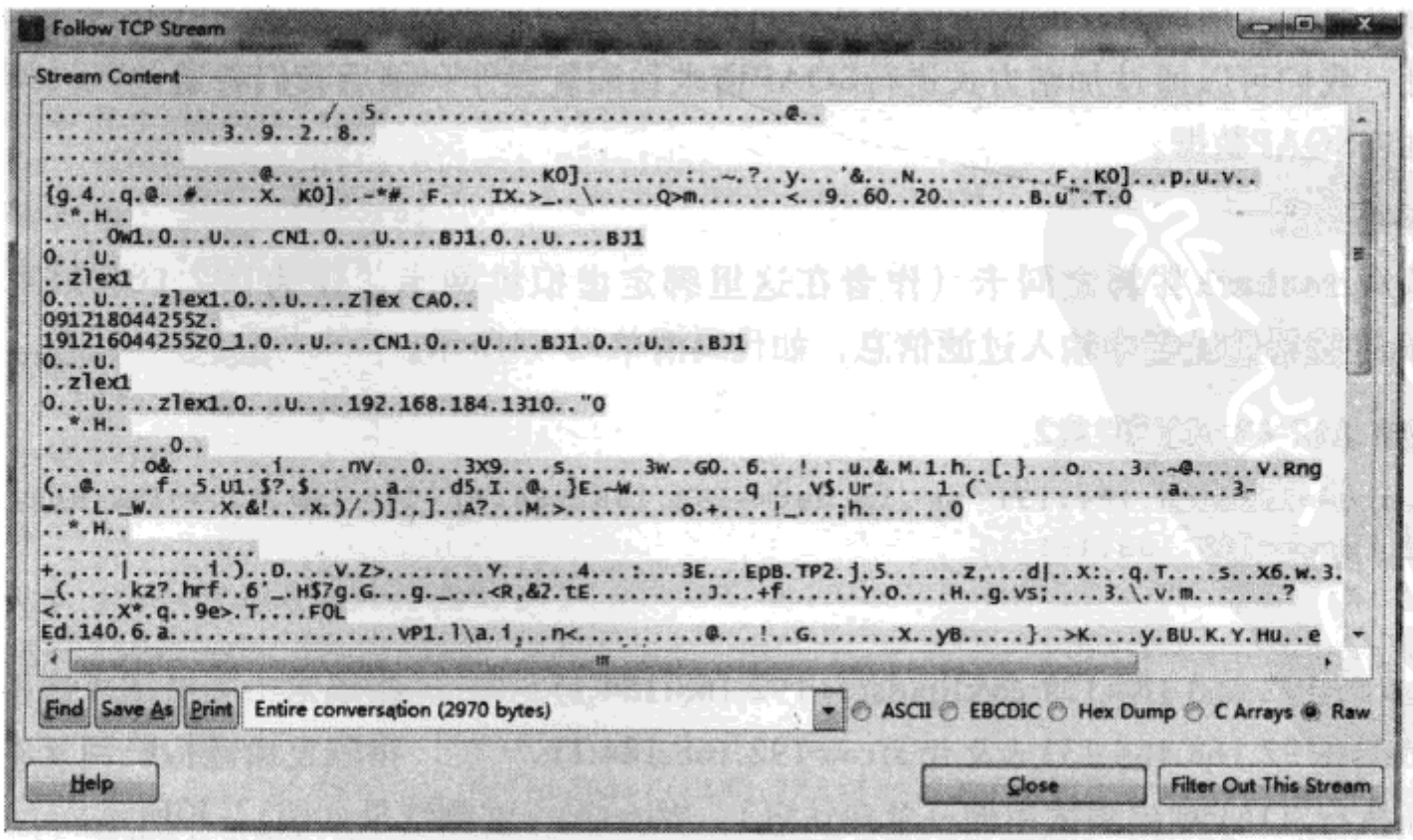


图12-31 SOAP加密交互内容

更多资源请访问稀酷客(www.ckook.com)

如果要在Wireshark中解析SSL/TLS交互内容，需要在Wireshark首选项（Preferences）中配置SSL协议。

单击Wireshark菜单“Edit”菜单项，在弹出的菜单中单击“Preferences”菜单项，如图12-32所示。

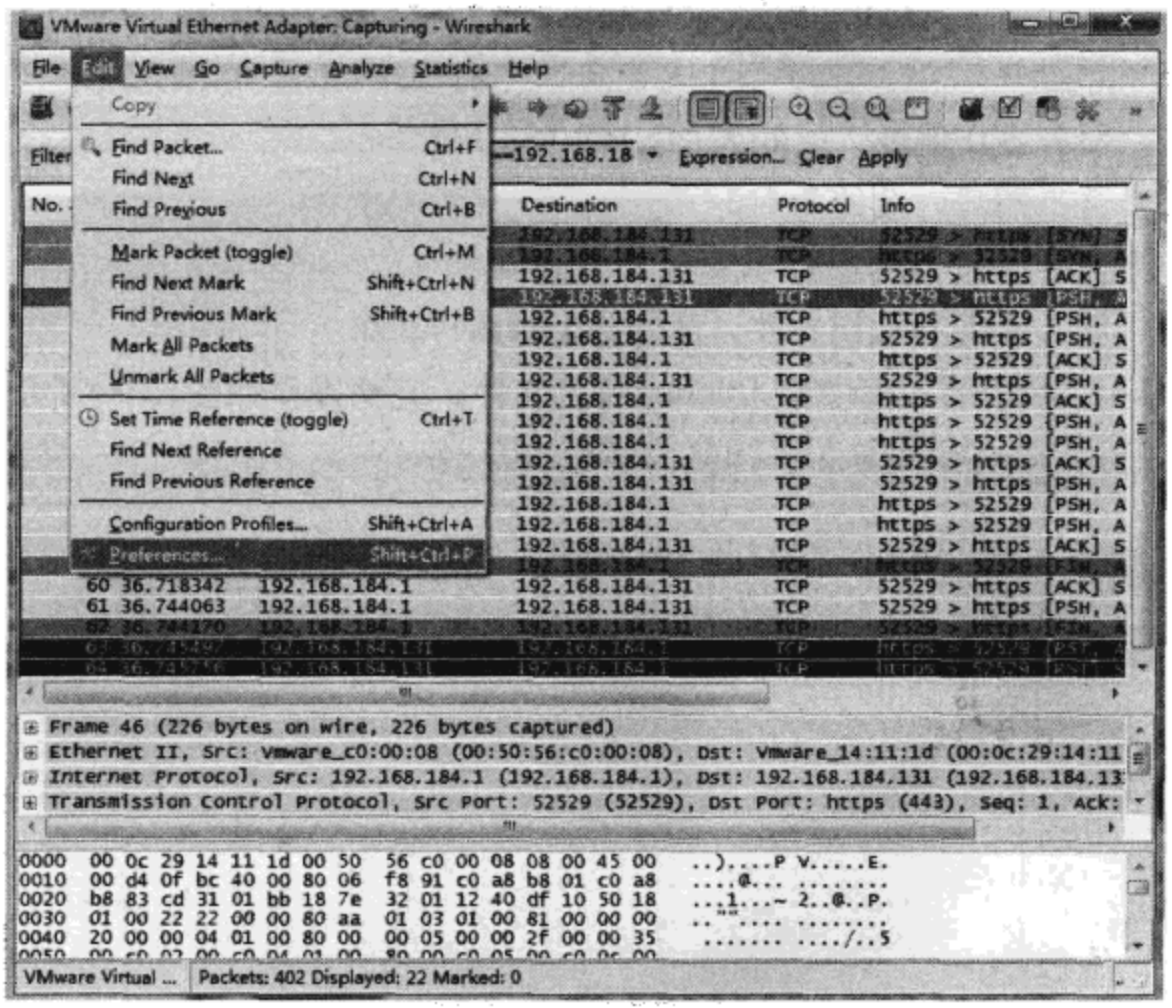


图12-32　Wireshark首选项菜单

打开Wireshark首选项对话框后，展开“Protocols”项，如图12-33所示。

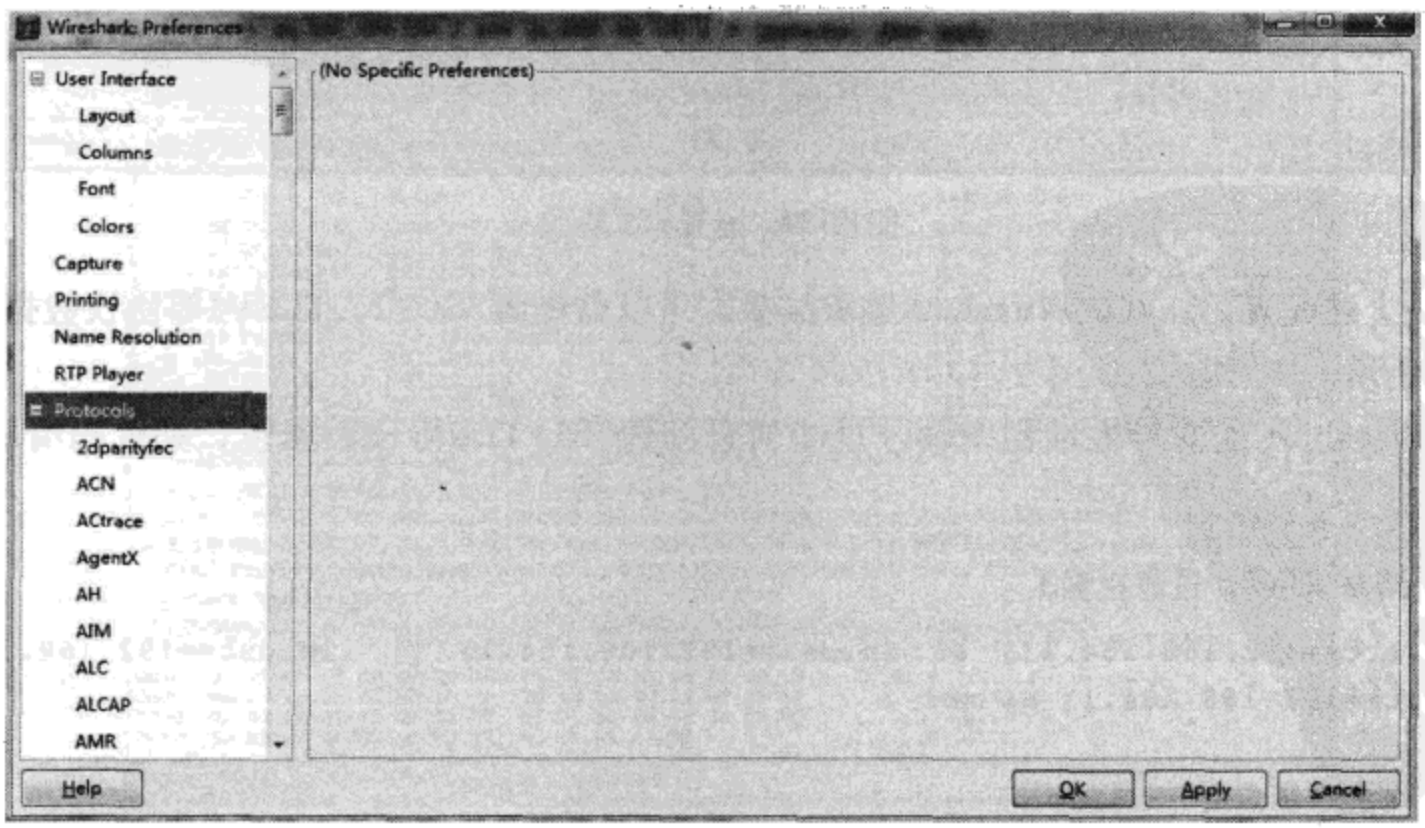


图12-33　Wireshark首选项对话框

更多资源请访问稀酷客(www.ckook.com)

在Protocols列表中选择“SSL”项，在右侧“RSA keys list”输入框中输入相关信息，输入格式如代码清单12-45所示。

代码清单12-45 RSA keys list输入格式

```
<ip>,<port>,<protocol>,<keyfile>
```

其中

<ip> 服务器IP地址，此处为192.168.184.131

<port> 监听端口，此处为443

<protocol> 监听协议，此处为https

<keyfile> 服务器证书私钥文件，此处为d:\serkey.pem

在“SSL debug file”输入框中输入debug日志文件。完整操作如图12-34所示。

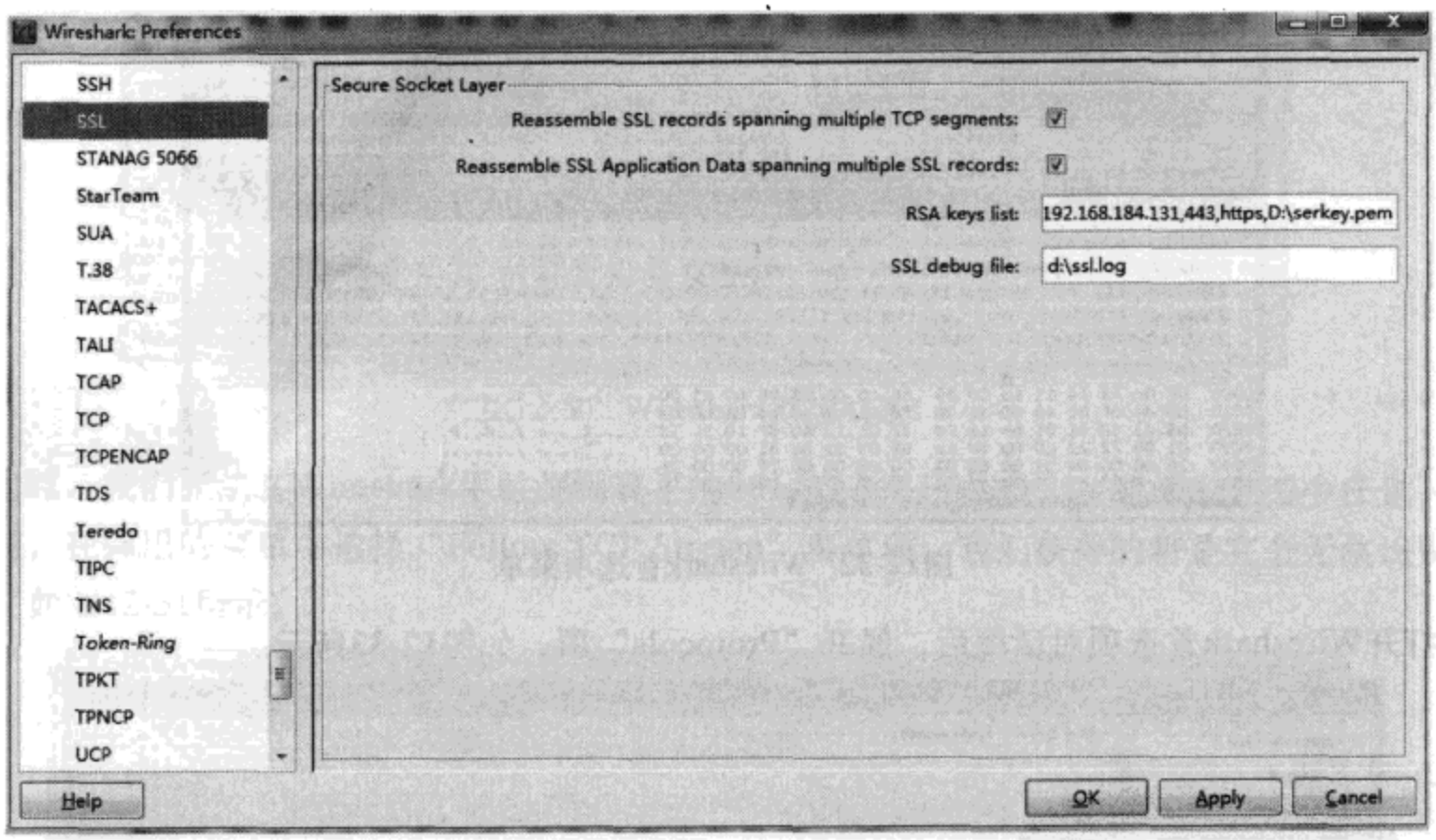


图12-34 配置SSL协议

完成上述配置后，观察Wireshark监测结果，可以监测到SSLV2、TLSV1等协议数据包，如图12-35所示。

在Wireshark过滤器地址栏中输入过滤信息筛选SSL/TLS协议数据包，如代码清单12-46所示。

代码清单12-46 过滤拦截3

```
(ip.src==192.168.184.131 && ip.dst==192.168.184.1) || (ip.dst==192.168.184.131
&& ip.src==192.168.184.1) && ssl
```

其中

(ip.src==192.168.184.131 && ip.dst==192.168.184.1) 指限定由本机请求虚拟机

更多资源请访问稀酷客(www.ckook.com)

(ip.dst==192.168.184.131 && ip.src==192.168.184.1)　　指限定由虚拟机回复本机

ssl　　指定SSL/TLS协议

执行上述修改后，Wireshark监测结果如图12-36所示。

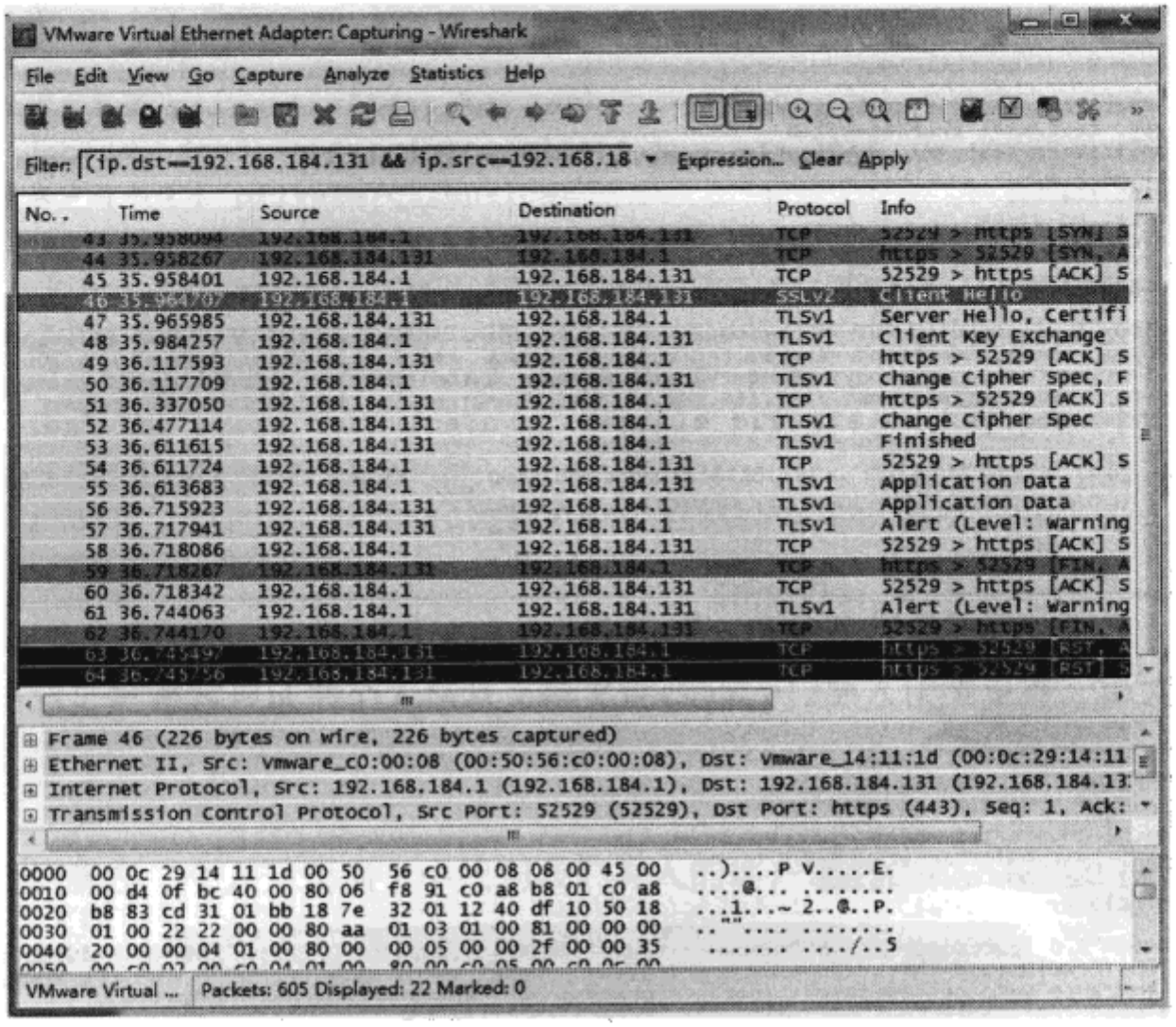


图12-35　Wireshark监测结果2

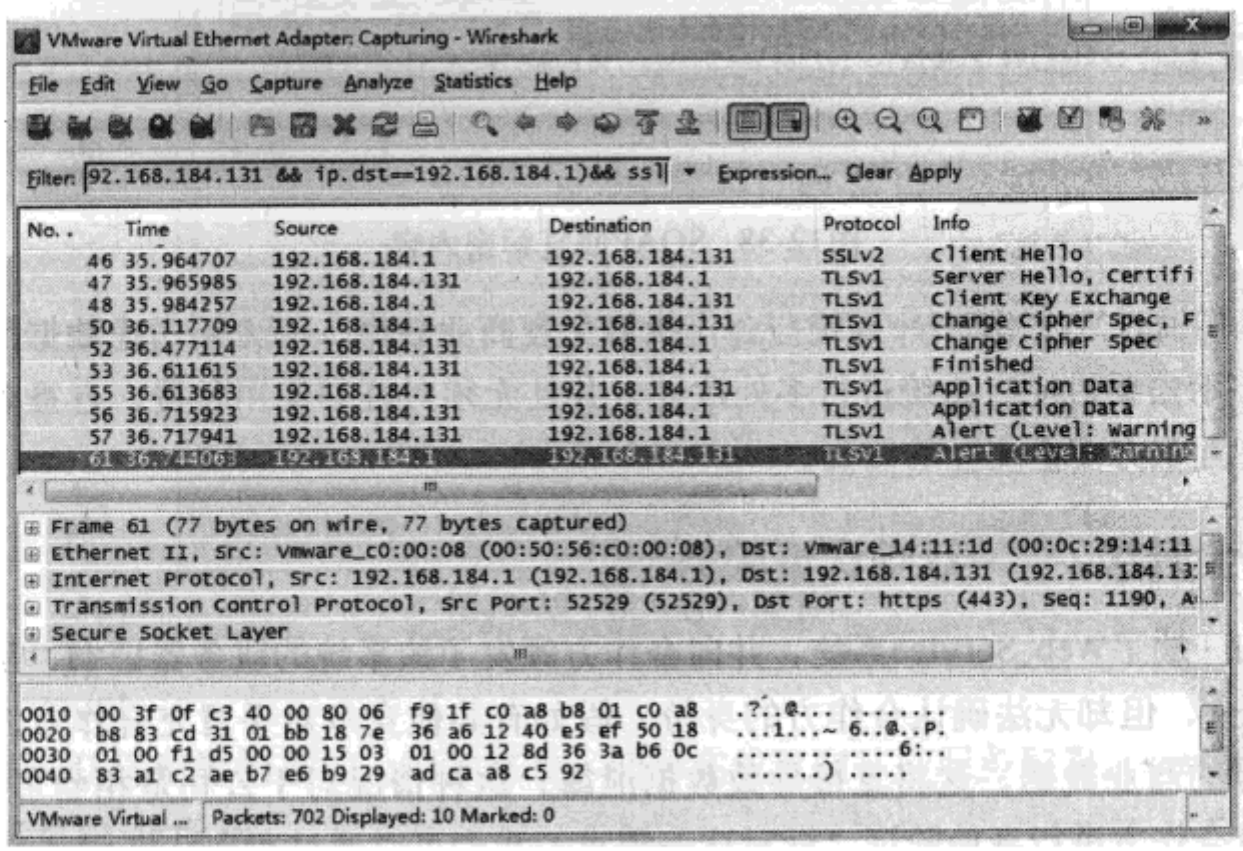


图12-36　Wireshark监测结果3

更多资源请访问稀酷客(www.ckook.com)

在图12-36中右键单击任意一条记录，在弹出的菜单中选择“Follow SSL Stream”菜单项。

在弹出的窗口（Follow TCP Stream）中单击下拉列表框选择请求和回复信息，分别得到SOAP请求和SOAP回复内容，如图12-37和图12-38所示。

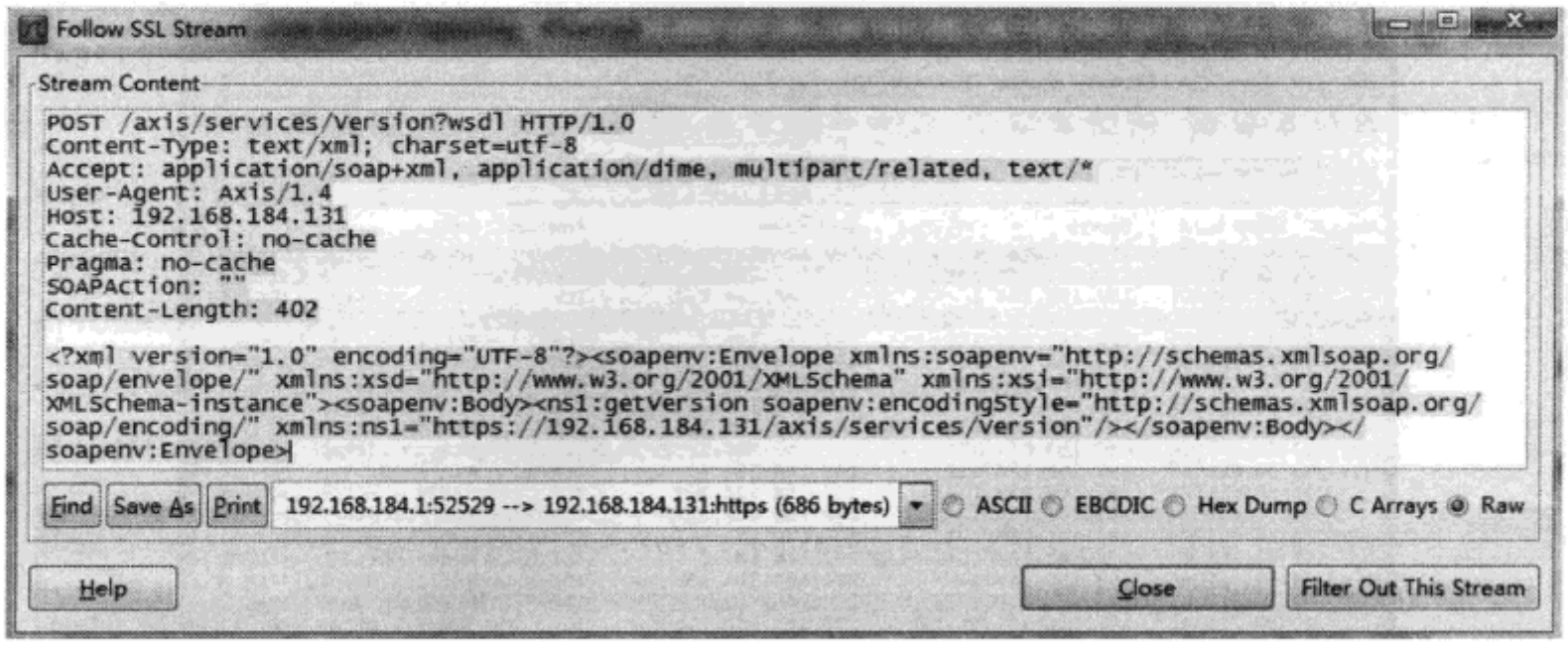


图12-37 SOAP请求解密内容

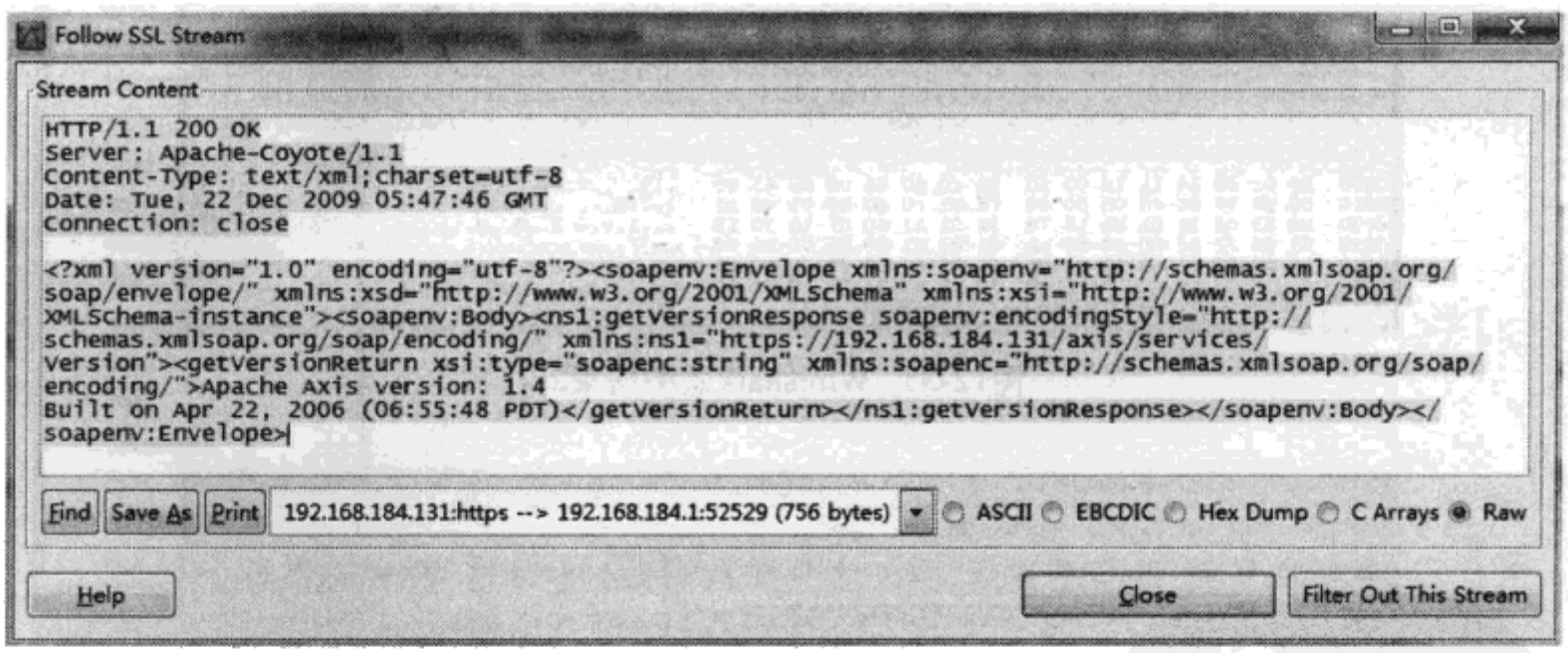


图12-38 SOAP回复解密内容

此时，我们通过Web Service系统与合作伙伴交互商业数据，而无须考虑数据泄露问题。数据交互双方并不需要对原有系统进行多处改动，并且无须对证书使用付费，自签名证书完全可以满足数据交互双方安全交互的需要。

12.3.3 安全升级2——双向认证服务

如果我们开放了Web Service系统，并向合作方颁发了该系统的服务器证书，虽然可以确保SOAP交互安全，但却无法确认合作方的身份。假如有其他竞争对手冒充合作方，则可以很轻易地获得我方的商业数据，这将直接导致数据泄露，这种情况对于公司是相当可怕的。这时，我们有必要对合作方进行身份验证。针对这一需求，我们可以通过双向认证服务满足。

更多资源请访问稀酷客(www.ckook.com)

1. 构建证书

在单向认证服务的基础上，我们构建客户证书用于验证客户身份。完成命令如代码清单12-47所示。

代码清单12-47　构建客户证书

```
echo 产生客户私钥 private/client.key.pem
openssl genrsa -des3 -out private/client.key.pem 2048
echo 生成客户证书请求 private/client.csr
openssl req -new -key private/client.key.pem -out
private/client.csr -subj "/C=CN/ST=BJ/L=BJ/O=zlex/OU=zlex/CN=zlex"
echo 签发客户证书 private/client.cer
openssl ca -in private/client.csr -days 3650 -out
certs/client.cer -cert certs/ca.cer -keyfile private/ca.key.pem -notext
echo 客户证书转换 private/client.p12
openssl pkcs12 -export -inkey private/client.key.pem -in certs/client.cer -out
certs/client.p12
```

❑ 导入证书

为方便演示，需将客户证书个人信息交换文件client.p12导入IE浏览器。请读者参照ca.p12文件导入方式导入client.p12文件，注意输入客户证书密码（这里为“123456”），并确认导入该证书。

最终，我们将在“个人”选项卡中找到我们导入的客户证书，如图12-39所示。

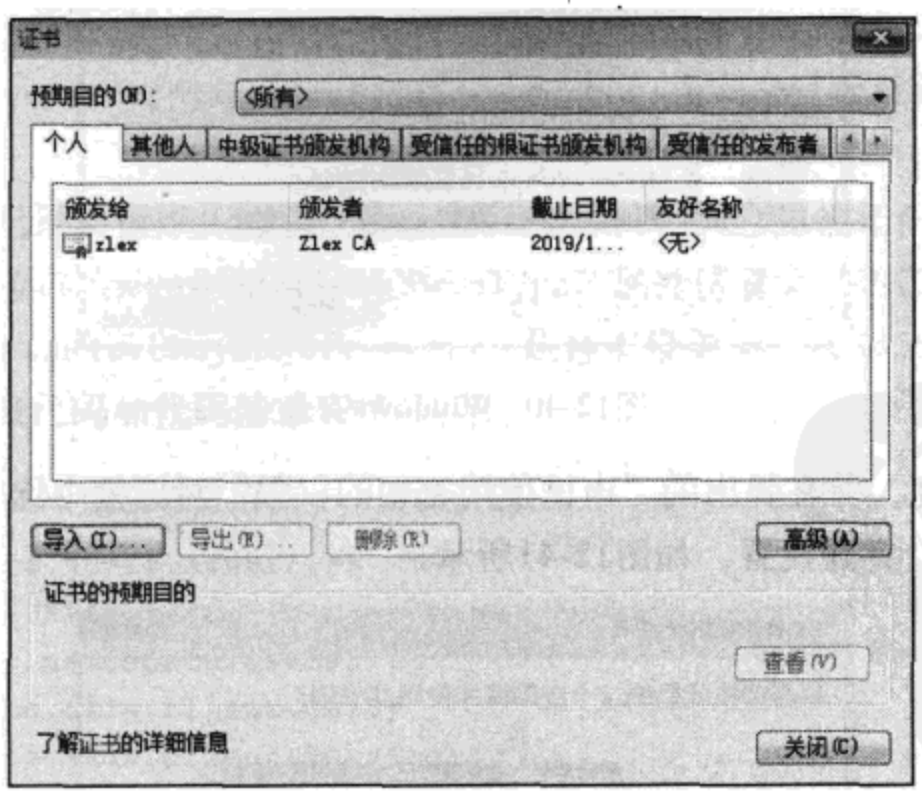


图12-39　导入客户证书

2. 配置Tomcat

我们在单向认证配置的基础上稍加修改，即可完成双向认证服务配置。server.xml文件详细配置如代码清单12-48所示。

更多资源请访问稀酷客(www.ckook.com)

代码清单12-48 server.xml配置——双向认证

```
<Connector
    clientAuth="true"
    SSLEnabled="true"
    maxThreads="150"
    port="443"
    protocol="HTTP/1.1"
    scheme="https"
    secure="true"
    sslProtocol="TLS"
    keystoreFile="conf/server.p12"
    keystorePass="123456"
    keystoreType="PKCS12"
    truststoreFile="conf/ca.p12"
    truststorePass="123456"
    truststoreType="PKCS12" />
```

这里我们需要将客户端验证开关打开，即将参数clientAuth值设置为“true”。

重新启动Tomcat，通过IP直接访问地址https://192.168.184.131/axis/services，我们将先得到“Windows安全”对话框，如图12-40所示。

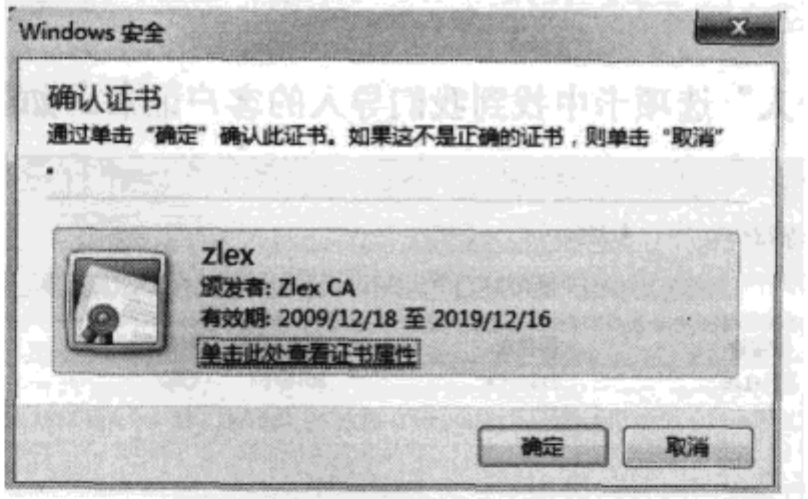


图12-40 Windows安全

点击“确定”按钮，并在弹出的“申请使用密钥的权限”对话框中选择“授予权限”选项，并单击“确定”按钮，完成设置，如图12-41所示。

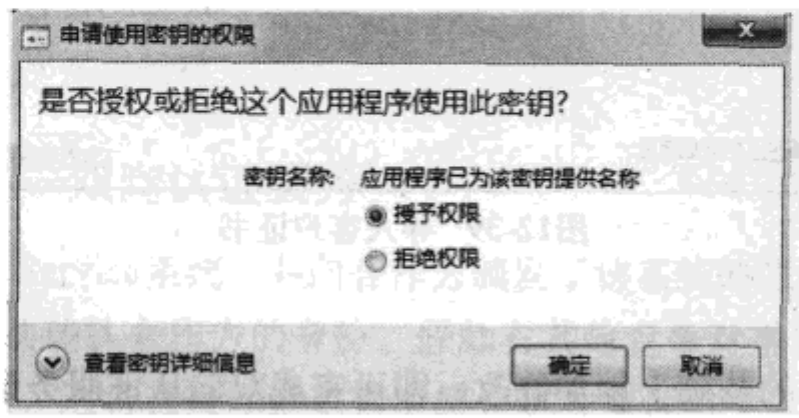


图12-41 申请使用密钥权限

更多资源请访问稀酷客(www.ckook.com)

最终，我们将得到服务列表，与图12-27一致。

3. 验证服务

构建双向认证服务，客户端需要导入客户证书，这里我们需要修改init()方法，同时初始化信任库配置和密钥库配置。

信任库相关属性请参见12.3.2节内容，密钥库相关属性详细描述如下：

- javax.net.ssl.keyStore：指向密钥库文件路径。
- javax.net.ssl.keyStoreType：密钥库文件类型。
- javax.net.ssl.keyStorePassword：密钥库密码。

此时，我们需要将服务器个人信息交换文件client.p12放置在D盘根目录下。初始化证书配置方法实现如代码清单12-49所示。

代码清单12-49　初始化证书配置——双向认证

```
// 初始化证书配置
@Before
public final void init() {
    // 配置信任库
    System.setProperty("javax.net.ssl.trustStore", "D:\\server.p12");
    System.setProperty("javax.net.ssl.trustStorePassword", "123456");
    System.setProperty("javax.net.ssl.trustStoreType", "PKCS12");
    // 配置密钥库
    System.setProperty("javax.net.ssl.keyStore", "D:\\client.p12");
    System.setProperty("javax.net.ssl.keyStoreType", "PKCS12");
    System.setProperty("javax.net.ssl.keyStorePassword", "123456");
}
```

此处我们仅需要配置密钥库，即将“javax.net.ssl.keyStore”属性指向文件client.p12，并指定密钥库文件类型，即将“javax.net.ssl.keyStoreType”属性值置为“PKCS12”。同时，指定密钥库密码，即将“javax.net.ssl.keyStorePassword”属性值设置为“123456”。

双向认证服务完整代码如代码清单12-50所示。

代码清单12-50　SOAP响应测试用例——双向认证

```
import static org.junit.Assert.*;
import java.net.URL;
import javax.xml.namespace.QName;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.junit.Before;
import org.junit.Test;
/**
 * WebService测试
 * @author 梁栋
 * @version 1.0
 * @since 1.0
```

更多资源请访问稀酷客(www.ckook.com)

```
 */
public class WebServiceTest {
    // Namespace URL
    private String namespaceUri = "https:// 192.168.184.131/axis/services/Version";
    // WSDL URL
    private String wsdlUrl = "https:// 192.168.184.131/axis/services/Version?wsdl";
    // 初始化证书配置
    @Before
    public final void init() {
            // 配置信任库
            System.setProperty("javax.net.ssl.trustStore", "D:\\server.p12");
            System.setProperty("javax.net.ssl.trustStorePassword", "123456");
            System.setProperty("javax.net.ssl.trustStoreType", "PKCS12");
            // 配置密钥库
            System.setProperty("javax.net.ssl.keyStore", "D:\\client.p12");
            System.setProperty("javax.net.ssl.keyStoreType", "PKCS12");
            System.setProperty("javax.net.ssl.keyStorePassword", "123456");
    }
    /**
     * 测试
     * @throws Exception
     */
    @Test
    public final void test() throws Exception {
            // 创建调用对象
            Service service = new Service();
            Call call = (Call) service.createCall();
            // 调用远程方法
            call.setOperationName(new QName(namespaceUri, "getVersion"));
            // 设置URL
            call.setTargetEndpointAddress(new URL(wsdlUrl));
            // 执行远程调用，同时获得返回值
            String version = (String) call.invoke(new Object[] {});
            // 打印信息
            System.err.println(version);
            // 验证
            assertNotNull(version);
    }
}
```

此时，我们可以通过加密方式进行SOAP请求和回复操作。同时，我们一样可以通过Wireshark监测并解析加密SOAP数据。

4. 网络监测

参照12.3.2节内容，对Wirdshark进行SSL/TLS协议配置，并在Wireshark过滤器地址栏中输入过滤信息筛选SSL/TLS协议数据包，如代码清单12-51所示。

更多资源请访问稀酷客(www.ckook.com)

代码清单12-51　过滤拦截4

```
(ip.src==192.168.184.131 && ip.dst==192.168.184.1) || (ip.dst==192.168.184.131
&& ip.src==192.168.184.1) && ssl
```

其中

(ip.src==192.168.184.131 && ip.dst==192.168.184.1)　　指限定由本机请求虚拟机

(ip.dst==192.168.184.131 && ip.src==192.168.184.1)　　指限定由虚拟机回复本机

ssl　　指定SSL/TLS协议

重新执行SOAP响应测试用例（双向认证），在Wireshark中监测到的SSL/TLS交互内容。右键单击任意一条数据包，在弹出的菜单中选择“Follow TCP Stream”菜单项。我们同样可以获得解密的SOAP交互内容，如图12-42所示。

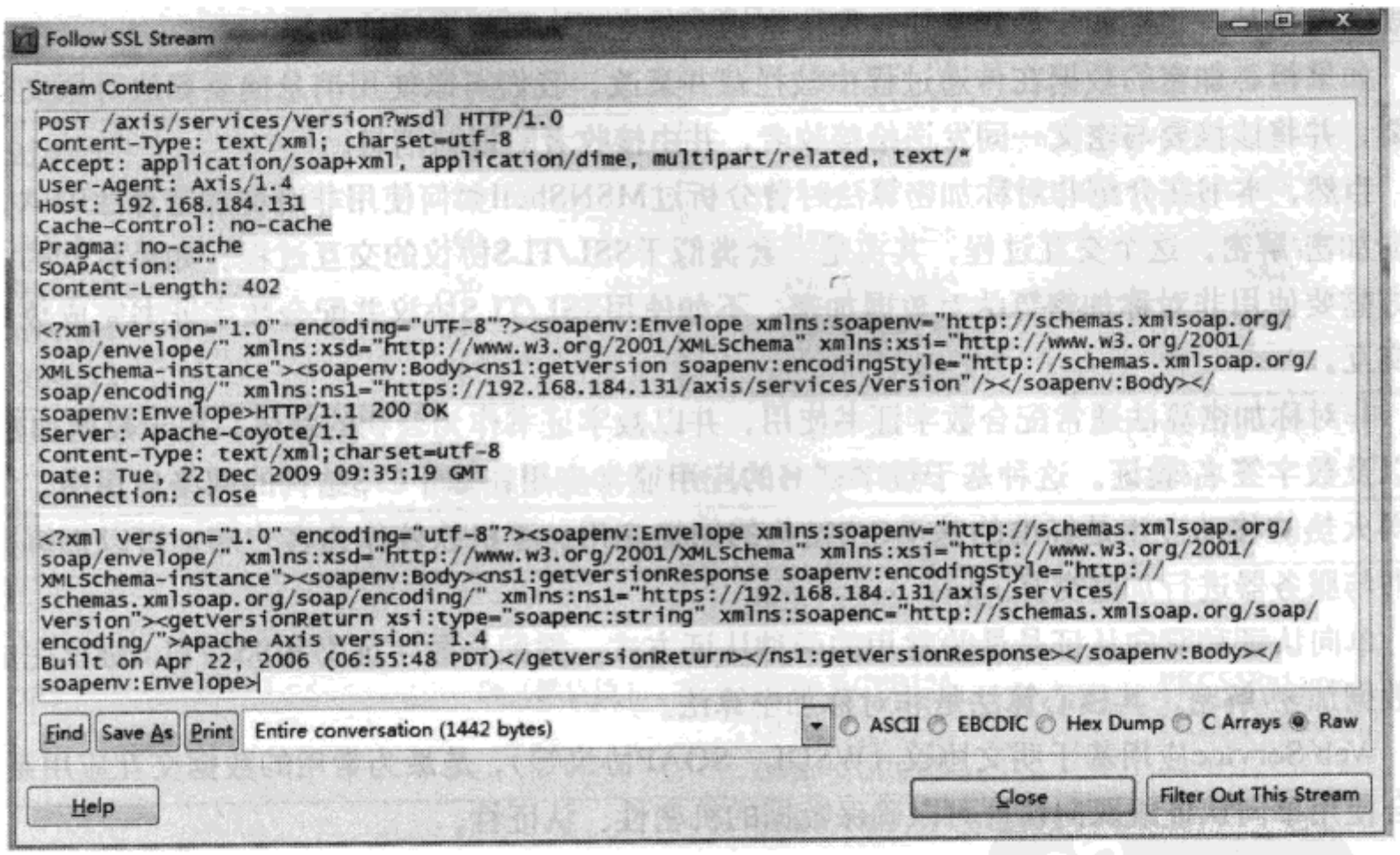


图12-42　SOAP交互内容

此处，我们使用了服务器证书私钥对上述内容解密，这说明在双向认证的网络交互中，服务器证书用于加密/解密，而客户证书仅用于验证客户身份。

12.4　小结

账号管理模块是众多系统中最重要的模块之一，用户密码则是这个模块中的最敏感的信息。倘若以明文方式将其存储在数据库或文件中，都极有可能成为泄露用户隐私的系统漏洞。

BBS、BLOG、SNS等B/S系统均以用户为核心，账号管理模块是这些系统中唯一的安全屏障。在设计这些系统时，我们需要注意合理隐蔽用户的私密信息，尽量避免敏感信息在系统中直接暴露。当用户遗失密码时，由系统根据随机信息重置密码，并以邮件方式告知用户新密码。

更多资源请访问稀酷客(www.ckook.com)

利用消息摘要算法的单向性，我们可以隐蔽用户的敏感信息。不同的摘要算法对应的摘要值长度可能有所不同，摘要值长度与系统安全性成正比，与系统的资源的消耗成反比。我们需要考虑摘要运算时间消耗、数据库存储空间消耗，以及系统改造所需的开发工作量，最终为系统选择合适的安全实现。

Base64算法经常被用来隐藏数据，虽然算法本身不具备安全性，但同样起到了数据隐藏的作用。如果我们对一些网络交互数据进行分析，会发现大部分数据实际上仅仅是做了简单Base64编码，电子邮件亦是如此。

通过对称加密算法，我们可以很方便地对聊天数据进行加密/解密。但是，仍有一个前提：我们需要将密钥妥善地交给进行聊天会话的交互双方，使其能够使用同一份密钥进行加密/解密。如果基于该密钥及算法的软件被破译，就意味着密钥的破解，这也是对称加密算法的软肋。常规的做法是，定期同步最新密钥，保持密钥不断更新，预防密钥被破解。

如果担心加密的数据在传递过程中被拦截并篡改，我们可以使用消息摘要算法对原文提取摘要，并将该摘要与密文一同发送给接收者，并由接收者验证该消息。

当然，本书在介绍非对称加密算法时曾分析过MSNShell如何使用非对称加密算法RSA完成数据加密/解密。这个交互过程，其实是一套类似于SSL/TLS协议的交互过程。如果数据的交互双方需要使用非对称加密算法对数据加密，不如使用SSL/TLS协议并配合数字证书完成数据加密交互。

非对称加密算法通常配合数字证书使用，并以数字证书作为公钥的载体，用于数据加密/解密以及数字签名/验证。这种基于数字证书的应用通常应用在基于C/S结构的网络应用中。如今炒得火热的移动应用是标准的基于C/S结构的网络应用。手机软件作为客户端，可以捆绑数字证书与服务器进行加密交互。

单向认证和双向认证是最为常用的两种认证方式，需配合数字证书和SSL/TLS协议完成底层数据加密/解密，其核心算法是非对称加密算法。

Web Service应用基于明文协议（WSDL、SOAP协议等），是最为常用的数据交互应用系统，配合使用单向认证或双向认证可以确保数据的机密性、认证性。

更多资源请访问稀酷客(www.ckook.com)

附录A

Java 6支持的算法

表A-1　Java 6支持的消息摘要算法

算法	摘要长度	算法	摘要长度
MD2	128	HmacMD5	128
MD5	128	HmacSHA1	160
SHA-1	160	HmacSHA256	256
SHA-256	256	HmacSHA384	384
SHA-384	384	HmacSHA512	512
SHA-512	512		

表A-2　Java 6支持的对称加密算法

算法	密钥长度	工作模式	填充方式
DES	56（默认值）	ECB，CBC，PCBC，CTR，CTS，CFB，CFB8至CFB128，OFB，OFB8至OFB128	NoPadding，PKCS5Padding，ISO10126Padding
DESede（Triple DES，3DES）	112，168（默认值）	ECB，CBC，PCBC，CTR，CTS，CFB，CFB8至CFB128，OFB，OFB8至OFB128	NoPadding，PKCS5Padding，ISO10126Padding
AES	128（默认值），192，256	ECB，CBC，PCBC，CTR，CTS，CFB，CFB8至CFB128，OFB，OFB8至OFB128	NoPadding，PKCS5Padding，ISO10126Padding
Blowfish	32至448（8的倍数）（默认值为128）	ECB，CBC，PCBC，CTR，CTS，CFB，CFB8至CFB128，OFB，OFB8至OFB128	NoPadding，PKCS5Padding，ISO10126Padding

更多资源请访问稀酷客(www.ckook.com)

（续）

算法	密钥长度	工作模式	填充方式
RC2	40至1024 （8的倍数） （默认值为128）	ECB	NoPadding
RC4（ARCFOUR）	40至1024 （8的倍数） （默认值为128）	ECB	NoPadding

注：在Java 6使用256位密钥的AES算法需要获得无政策限制权限文件（Unlimited Strength Jurisdiction Policy Files）。

表A-3 Java 6支持的对称加密算法——PBE系列

算法	密钥长度	工作模式	填充方式
PBEWithMD5AndDES	56（默认值）	CBC	PKCS5Padding
PBEWithMD5AndTripleDES	112， 168（默认值）	CBC	PKCS5Padding
PBEWithSHA1AndDESede	112， 168（默认值）	CBC	PKCS5Padding
PBEWithSHA1AndRC2_40	40至1024 （8的倍数） （默认值为128）	CBC	PKCS5Padding

表A-4 Java 6支持的非对称加密算法

算法	密钥长度	工作模式	填充方式
DH（Diffie-Hellman）	512至1024位（64的倍数） （默认值为1024）	—	—
RSA	512至65536位（64的倍数） （默认值为1024）	ECB	NoPadding， PKCS1Padding， OAEPWITHMD5AndMGF1Padding， OAEPWITHSHA1AndMGF1Padding， OAEPWITHSHA256AndMGF1Padding， OAEPWITHSHA384AndMGF1Padding， OAEPWITHSHA512AndMGF1Padding

表A-5 Java 6支持的数字签名算法

算法	密钥长度	签名长度
MD2withRSA MD5withRSA SHA1withRSA	512至65536位（64的倍数） （默认值为1024）	与密钥长度相同
SHA1withDSA	512至1024位（64的倍数） （默认值为1024）	—

更多资源请访问稀酷客(www.ckook.com)

附录B

Bouncy Castle支持的算法

表B-1 Bouncy Castle支持的消息摘要算法

算法	摘要长度	算法	摘要长度
MD2	128	SHA-512	512
MD4	128	HmacMD2	128
MD5	128	HmacMD4	128
RipeMD128	128	HmacRipeMD128	128
RipeMD160	160	HmacRipeMD160	160
RipeMD256	256	HmacSHA224	224
RipeMD320	320	Hmac-Tiger	192
SHA-1	160	Tiger	192
SHA-224	224	GOST3411	256
SHA-256	256	Whirlpool	512
SHA-384	384		

表B-2 Bouncy Castle支持的对称加密算法——DES

算法	密钥长度	工作模式	填充方式
DES	56（默认值），64	ECB，CBC，PCBC，CTR，CTS，CFB，CFB8至CFB128，OFB，OFB8至OFB128	PKCS7Padding，ISO10126d2Padding，X932Padding，ISO7816d4Padding，ZeroBytePadding
DESede（Triple DES，3DES）	128，168（默认值），192	ECB，CBC，PCBC，CTR，CTS，CFB，CFB8至CFB128，OFB，OFB8至OFB128	PKCS7Padding，ISO10126d2Padding，X932Padding，ISO7816d4Padding，ZeroBytePadding
AES	128（默认值），192，256	ECB，CBC，PCBC，CTR，CTS，CFB，CFB8至CFB128，OFB，OFB8至OFB128	PKCS7Padding，ZeroBytePadding

更多资源请访问稀酷客(www.ckook.com)

（续）

算法	密钥长度	工作模式	填充方式
IDEA	128（默认值）	ECB	PKCS5Padding， PKCS7Padding， ISO10126Padding， ZeroBytePadding

表B-3 Bouncy Castle支持的对称加密算法——PBE系列

算法	密钥长度（默认值）	工作模式	填充方式
PBEWithMD2AndDES	64	CBC	PKCS5Padding， PKCS7Padding， ISO10126Padding， ZeroBytePadding
PBEWithMD2AndRC2	128	CBC	PKCS5Padding， PKCS7Padding， ISO10126Padding， ZeroBytePadding
PBEWithMD5AndDES	64	CBC	PKCS5Padding， PKCS7Padding， ISO10126Padding， ZeroBytePadding
PBEWithMD5AndRC2	128	CBC	PKCS5Padding， PKCS7Padding， ISO10126Padding， ZeroBytePadding
PBEWithSHA1AndDES	64	CBC	PKCS5Padding， PKCS7Padding， ISO10126Padding， ZeroBytePadding
PBEWithSHA1AndRC2	128	CBC	PKCS5Padding， PKCS7Padding， ISO10126Padding， ZeroBytePadding
PBEWithSHAAnd2-KeyTripleDES-CBC	128	CBC	PKCS5Padding， PKCS7Padding， ISO10126Padding， ZeroBytePadding
PBEWithSHAAnd3-KeyTripleDES-CBC	192	CBC	PKCS5Padding， PKCS7Padding， ISO10126Padding， ZeroBytePadding

更多资源请访问稀酷客(www.ckook.com)

（续）

算法	密钥长度（默认值）	工作模式	填充方式
PBEWithSHAAnd128BitRC2-CBC	128	CBC	PKCS5Padding， PKCS7Padding， ISO10126Padding， ZeroBytePadding
PBEWithSHAAnd40BitRC2-CBC	40	CBC	PKCS5Padding， PKCS7Padding， ISO10126Padding， ZeroBytePadding
PBEWithSHAAnd128BitRC4	128	CBC	PKCS5Padding， PKCS7Padding， ISO10126Padding， ZeroBytePadding
PBEWithSHAAnd40BitRC4	40	CBC	PKCS5Padding， PKCS7Padding， ISO10126Padding， ZeroBytePadding
PBEWithSHAAndTwofish-CBC	256	CBC	PKCS5Padding， PKCS7Padding， ISO10126Padding， ZeroBytePadding
PBEWithSHAAndIDEA-CBC	128	CBC	PKCS5Padding， PKCS7Padding， ISO10126Padding， ZeroBytePadding

表B-4 Bouncy Castle支持的非对称加密算法

算法	密钥长度	工作模式	填充方式
RSA	512至65536位（64的倍数） （默认值为2048）	NONE	NoPadding， PKCS1Padding， OAEPWithMD5AndMGF1Padding， OAEPWithSHA1AndMGF1Padding， OAEPWithSHA224AndMGF1Padding， OAEPWithSHA256AndMGF1Padding， OAEPWithSHA384AndMGF1Padding， OAEPWithSHA512AndMGF1Padding， ISO9796-1Padding

更多资源请访问稀酷客(www.ckook.com)

（续）

算法	密钥长度	工作模式	填充方式
ElGamal	160至16384位（8的倍数） （默认值为1024）	ECB， NONE	NoPadding， PKCS1Padding， OAEPWithMD5AndMGF1Padding， OAEPWithSHA1AndMGF1Padding， OAEPWithSHA224AndMGF1Padding， OAEPWithSHA256AndMGF1Padding， OAEPWithSHA384AndMGF1Padding， OAEPWithSHA512AndMGF1Padding， ISO9796-1Padding

表B-5 Bouncy Castle 支持的数字签名算法

算法	密钥长度	签名长度
SHA224withRSA SHA256withRSA SHA384withRSA SHA512withRSA RIPEMD128withRSA RIPEMD160withRSA	512至65536位（64的倍数） （默认值为2048）	与密钥长度相同
SHA224withDSA SHA256withDSA SHA384withDSA SHA512withDSA	512至1024位（64的倍数） （默认值为1024）	—
NONEwithECDSA	—	128
RIPEMD160withECDSA		160
SHA1withECDSA		160
SHA224withECDSA		224
SHA256withECDSA		256
SHA384withECDSA		384
SHA512withECDSA		512

更多资源请访问稀酷客(www.ckook.com)